

# USING C++ AS A SCIENTIFIC PROGRAMMING LANGUAGE

James S. Peery, Kent G. Budge, Allen C. Robinson,  
*Computational Physics Research and Development Division*  
*Sandia National Laboratories*  
*Albuquerque, New Mexico*

and

David Whitney  
*Cray Research*  
*Eagan, Minnesota*

## 1. Introduction

Large computational physics codes are increasing in complexity as customers demand improved physics packages and more flexible algorithms and problem specifications. It is not uncommon for a code to exceed one hundred thousand lines of FORTRAN, and some codes are much larger. This poses a considerable challenge for program management.

The Computational Physics Research and Development Division at Sandia National Laboratories is aggressively pursuing C++ as the language of choice for new coding efforts. We feel that we cannot meet the stringent customer requirements and delivery schedules we now face with either FORTRAN77 or Fortran-90.

### 1.1 General Advantages and Disadvantages of C++

The advantages of C++ are:

*Strong type checking.* All variables and procedures are declared prior to their first use. The argument lists of procedures must match in separate compilation units. These strong type checking features eliminate entire classes of program bugs common in FORTRAN programs.

*Superior language syntax.* C++ is a superset of ANSI C and shares its inherently block-structured syntax. The language is free format in the sense that few assumptions are made about white space. Thus, no artificial constraints on line length and continuation are imposed.

*Object-oriented programming support.* The definitive feature of C++ is the `class`, which is an extension of the `struct` feature of C. A `class` consists of a set of data members and an enumeration of functions that have access to those data members. It thus enforces the concept of data encapsulation. Furthermore, it is possible to invoke the functions associated with a class object without knowing the exact type of the object; this supports polymorphism.

*Superior dynamic memory management.* C++ provides strong, flexible support for the management of the free memory store. This is vital for codes operating on large and flexible databases.

*Operator and function overloading.* C++ permits the meaning of the standard operators to be redefined when applied to class objects. This permits unusually transparent programming syntax which is tailor-made for particular applications.

*Availability.* C++ is available or is relatively easy to port to any platform that supports C.

The disadvantages of C++ are:

*Lack of an ANSI language standard.* The language is very young and is still evolving. It will be some time before the ANSI standard is completed. Fortunately, the AT&T C<sub>FRONT</sub> translator is a *de facto* standard that is largely followed by other implementations.

*Poor optimization.* C++ code can be very inefficient. Many of the known efficiency issues can be addressed by careful coding. Others will be resolved only by the development of more sophisticated compilers. We believe that C++ can ultimately become competitive with FORTRAN from an efficiency standpoint on all computer architectures.

### 1.2 The Rising Popularity of C++

C++ is growing in popularity as a scientific programming language. At Sandia National Laboratories, the next generation shock wave physics code, RHALE++<sup>1</sup> and fluid dynamics code, ZEPHYR++<sup>2</sup> are being developed in C++. In addition, PCTH<sup>3</sup>, the parallel version of the 3-D Eulerian shock wave physics code, CTH, is being written in C++ and will run on SIMD and MIMD architectures. Moreover, Sandia development groups are using the third party product ACIS<sup>4</sup> which is written in C++, to provide solid modeling capabilities for their finite element codes.

In addition to Sandia, C++ scientific programming activities exist at Northrup Corporation<sup>5</sup>, University of New Mexico<sup>6</sup>, University of Colorado<sup>7</sup>, and Los Alamos<sup>8</sup>.

## 2. C++: A Language for Mathematical Physics

C++ can be regarded as a meta-language whose dialects are portable from one platform to another. Therefore, one can tailor a particular dialect to the expression of concepts in mathematical physics. Through this approach, one is capable of writing code that resembles the original equations. In other words, it is possible to write code that “reads like a book.” For example, given the equation

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

the corresponding C++ coding is written as

$$f = \text{Div}(T) + b;$$

where  $f$  and  $b$  are instances of a vector field class and  $T$  is an instance of a tensor field class. It should be noted that the language does not implicitly provide the means for adding vectors but rather gives a programmer all the tools necessary to define how the mathematics should be performed.

This example points out the difference between a “user” programmer and a “library” programmer. A user programmer would program the physics as demonstrated above with Equation 1. It is the library programmer’s responsibility to develop vector and tensor classes that encapsulate the data and functions necessary to describe the mathematics. For example, a vector class could be defined with data and member functions as

```
class Vector{
    double x, y, z;
public
    Vector();
    Vector(double x, double y, double z);
    ~Vector;
    Vector& operator=( );
    Vector operator+(Vector&);
};
```

The coding for the overloaded addition operator could be written as

```
Vector Vector::operator+(Vector& b)
{
    Vector c;
    c.x = x + b.x
    c.y = y + b.y
    c.z = z + b.z
    return c;
}
```

Hiding the actual operations in this fashion not only provides a simple interface for user code but also isolates many coding errors and architecture dependencies.

Programs that model physics typically require many types of fields: scalar, vector, and tensor fields. For example, a finite element code modeling continuum mechanics would require scalar, vector, and tensor fields to describe pressure, displacement, and stress for each element. By developing rich class libraries for these fields that reflect consistent mathematical definitions, a user programmer can simply develop his program by typing in equations. Through the strong type checking feature of the language, any inconsistencies in the expressions will be flagged by the compiler. In addition, these field classes hide the bookkeeping and indices from the user programmer, thus eliminating an entire class of bugs.

In order to illustrate the power of the C++ language, consider the divergence of the stress term in Equation 1. The physicist who wants to model this phenomena is most interested in the internal force field and is least interested in the topology of the problem. If the physicist is given symmetric tensor fields defined for pressure points (element centers) and vector fields defined at displacement points (element vertices), he can find the internal force field by rotating the stress to the current configuration and finding its divergence. This is given as

```
VectorField BlockSpec::InternalForce()
{
    // Rotate stress to current configuration

    SymTensorField Rotated_Stress =
        Sym(Rotation * Stress *
            Trans(Rotation));

    // Calculate and return the internal forces

    VectorField InForce = Div(Rotated_Stress,
        CurCoord);

    return InForce;
}
```

This sort of code is easy for a physicist to understand and to modify. The library classes themselves are “black boxes” so far as the physicist is concerned; he very rarely will need to know anything about their internal workings. Note, that the above example does not provide any information about the dimensionality or topology of problem; it simply replicates a portion of Equation 1. Of course, the language cannot prevent bugs introduced by incorrect equations. In the example above, it would be syntactically correct to find the divergence of the unrotated stress and thus always calculate the initial internal force instead of the current internal force.

### 3. C++ Efficiency Issues

Programming in terms of objects is not without its drawbacks. C++ implementations are notorious for inefficiencies in memory and execution speed. This is particularly true for scientific programming since overloading operators will be desired. In these types of member functions, C++ is a “memory hog.” For example, consider a matrix class with overloaded + and \* operators and the following expression:

```
Matrix B(rows, cols);
Matrix C(rows, cols);
Matrix D(rows, cols);
Matrix A;
```

$$A = B + C * D$$

where A, B, and C are matrices. Using overloaded operators, this expression will result in the following set of calls to evaluate the expression

```
Matrix Matrix::operator*(Matrix& )
    Matrix::Matrix(int& rows, int& cols)
    Matrix::Matrix(Matrix &)
    Matrix::~~Matrix()Matrix
Matrix::operator+(Matrix& )
    Matrix::Matrix(int& rows, int& cols)
    Matrix::Matrix(Matrix &)
    Matrix::~~Matrix()
Matrix& Matrix::operator=(Matrix&)
    Matrix::~~Matrix()
```

which in pseudo code is

```
operator *
    1) create temp_1 of size rows by cols [4]
    2) temp_1 = C * D [4]
    3) temp_2 = temp_1 [5]
    4) delete temp_1 [4]
operator +
    5) create temp_3 of size rows by cols [5]
    6) temp_3 = B + temp_2 [5]
    7) temp_4 = temp_3 [6]
    8) delete temp_3 [5]
operator =
    9) create A of size rows by cols [6]
    10) A = temp_4 [6]
    11) delete temp_2 [5]
    12) delete temp_4. [4]
```

where the number in brackets is the number of allocated matrices at the corresponding step. Without user optimization, steps 3, 7 and 10 require a loop over the total length of the matrix where each element of a matrix is assigned the value of another matrix

element. This is a tremendous number of unnecessary assignment operations. In addition, while the user only requested four matrices, at steps 7, 9, and 10 there are six matrices allocated. In general, every overloaded operator on the right hand side of an expression requires a temporary. For large memory objects, the creation of temporaries could severely limit the complexity of expressions. Later it will be shown that allocating memory can be very expensive in terms of CPU cycles and therefore, the creation of temporaries should be avoided.

We have implemented two methods to control the creation of temporaries: reference counting and container classes.

#### 3.1 Reference Counting

Reference counting basically eliminates the creation of objects in copy constructors and overloaded = operator. Reference counting can be implemented in a class by adding an integer pointer to the private data of a class. With reference counting, the pseudo code for the matrix class example becomes

```
operator *
    1) create temp_1 of size rows by cols [4]
    allocate temp_1.rc; set to zero
    2) temp_1 = C * D [4]
    3) &temp_2 = &temp_1; *temp_1.rc++ [4]
    4) *temp_1.rc-- [4]
operator +
    5) create temp_3 of size rows by cols [5]
    allocate temp_1.rc; set to zero
    6) temp_3 = B + temp_2 [5]
    7) &temp_4 = &temp_3; *temp_3.rc++ [5]
    8) temp_3.rc--; [5]
operator =
    9) &A = &temp_4 (= &temp_3) [5]
    *temp_4.rc++ (= *temp_3.rc)
    10) delete temp_2 [4]
    11) *temp_4.rc-- (= *temp_3.rc = A.rc) [4]
```

where rc is the integer pointer added to the matrix class. As one can see in this example, reference counting eliminated the unnecessary assignment operations but only eliminated one temporary. Reference counting will only eliminate one temporary per expression and thus it would be more memory efficient to program the matrix expression as

```
A = C * D;
A += B;
```

which leads to the pseudo code

```
operator *
    1) create temp_1 of size rows by cols [4]
    allocate temp_1.rc; set to zero
    2) temp_1 = C * D [4]
```

```

3) &temp_2 = &temp_1; *temp_1.rc++ [4]
4) *temp_1.rc-- [4]
operator =
5) &A = &temp_2 [4]
   *temp_2.rc++ (= *temp_1.rc = A.rc)
6) *temp_2.rc-- (= *temp_1.rc = A.rc) [4]
operator +=
7) A += B. [4]

```

The programming style shown above eliminates temporaries and unnecessary assignment operations.

In order to test the efficiency of C++ with reference counting, a matrix test case was constructed that basically performs a series of matrix operations. This test case does not resemble any known algorithm. Equivalent coding was developed in FORTRAN. A set of matrices with rank 95 were used and the results from the two codes are given in Tables 1 and 2 for different compiler options on a SUN SparcStation II and a single processor on a CRAY Y-MP.

**Table 1: C++ Matrix Class Test 95 x 95**

Vendor	Options	User Time	Ratio CPU / Peak CRAY CPU
SUN	(default)	6.0	26.5
SUN	-O4	3.4	14.8
CRAY	(default)	1.53	6.7
CRAY	-O1	1.52	6.6
CRAY	-O2	1.42	6.2
CRAY	-h vector0	1.52	6.6
CRAY	-h restrict=f	1.39	6.0
CRAY	-h restrict, bl	1.26	5.5
CRAY	-h ivdep	0.24	1.0
CRAY	-h ivdep, bl	0.23	1.0

Table 1 shows a vectorization gain of 6.6 for the C++ coding executed on the CRAY (-h vector0 versus -h ivdep). However, when one compares the SUN execution speed to the CRAY, it is evident that the C++ coding is not even approaching the peak speeds of the CRAY Y-MP. In addition, the performance of C++ is even more disturbing when one compares the execution speeds of FORTRAN to C++ for both architectures. As shown in Table 2, the FORTRAN code actually ran almost five times faster than the

equivalent C++ coding on the CRAY (C++ is more than 30 times slower, if C++ default compilation is compared to FORTRAN

**Table 2: Equivalent FORTRAN Test 95 x 95**

Vendor	Options	User Time	User Time Ratio F/C++ (C++/F)
SUN	-O4	0.9	0.26 (3.8)
CRAY		0.049	0.21 (4.8)

default compilation). Further investigation revealed that the discrepancy in these numbers could be partly explained in two observations: (1) chaining of functional units cannot be obtained with the C coding generated by AT&T CFRONT and, (2) a tremendous number of cycles are lost in allocating memory from the heap: a requirement in overloaded operator functions. Later, other observations determined that the parts of C++ coding that did vectorize, only vectorized for a maximum loop size of 64. The compiler option -h restrict=f informs the compiler that pointer arguments passed through the function call are not aliased elsewhere. With a matrix class, the pointer argument is an instance of a matrix object and not a pointer to the data. It would be very beneficial if C++ would allow the keyword "restrict" (an extension to ANSI C provided by CRAY to promote full vectorization of loop constructs in C coding). This limitation can be bypassed if C functions are called to perform the actual matrix operations. Results from this enhancement are given in the next section.

Currently, we do not know of a method in the current C++ language definition to obtain chaining when overloaded operators are used. For chaining to work with these types of functions, the CRAY C compiler would not only have to combine loops but also eliminate the memory allocation and deallocation calls that appear between the loops for temporary object creation and destruction. Reference counting helps these operations to be memory efficient but it does not eliminate many of operations that occur between the loops. If C++ allowed for overloading ternary operators, chaining could be achieved.

### 3.2 Container Classes

In the previous section it was argued that part of the inefficiencies in C++ lie with allocating and deallocating memory for the data segment of temporary objects. One method of eliminating these inefficiencies is to develop a class that manages its own data. Classes that manage themselves are called container classes. The general rule of container classes is to never deallocate memory that has been allocated. For example, in a series of operations, temporary objects are created and deleted. Most of the inefficiency in this process results from user specified data segment allocation and deallocation (we are not concerned with the allocation

and deallocation of an instance of an object). Within a container class, when memory is needed for the data segment of the class, the class first looks at its “free store” to see if memory is already allocated. If not, the new object allocates memory from the heap. When an object is deleted, its data segment is returned to the “free store” for the class. At some point in the execution of the program, the maximum amount of memory required for the problem is reached and the container class will cease to request memory from the heap.

In order to test the effects of the container class concept on the efficiency of C++, a very simple test case was developed and is shown below.

```
Matrix a(len);
Matrix b(len);
Matrix c;
for(register i=0;i < max_iterations; i++) {
    c = a + b;
}
```

This test case was programmed in FORTRAN, C, and C++. The test case was intentionally kept simple to determine if C++ could be made as efficient as FORTRAN. Since C++ is translated to C by AT&T CFRONT before it is compiled, equivalent C coding was developed to determine the optimization that could be achieved with the CRAY C compiler on the mangled code that AT&T CFRONT produces. In addition, many variations of loop constructs were tested in the three languages. The megaflop (Mflops) performances of the three different codes on the CRAY are given in Table 3. Each code was compiled with the most aggressive compiler optimizations available.

**Table 3: Matrix Test a = b + c (rank = 100, 10,000 iterations)**

Code	Mflops	Remarks
Fortran	58.3	Two loops, outer index - inner loop
Fortran	102.21	Two loops, inner index - inner loop
Fortran	143.9	Single loop
C	92.9	Single loop
C	135.2	Declare argument pointers as restrict
C++	38.3	No memory management - Only reference counting.
C++	90.3	Use a.Add(b,c) function - no temporaries are required.
C++	84.1	Memory management and reference counting
C++	121.5	Memory management, reference counting and call to a C function with argument pointers declared as restrict

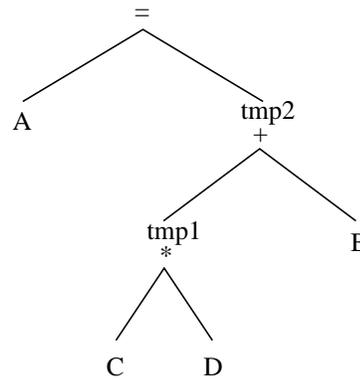
Table 3 shows that this particular C++ coding performed well against FORTRAN and C coding under the conditions that one develops classes that use reference counting, perform their own memory management and take advantage of optimizations provided by the vendor. Note that by developing code in C++, when an optimization is made in a class function, the optimization is realized in every portion of the code that uses the member function. In other words, the user code does not have to be modified. In a FORTRAN code, the optimization would have to be replicated in every location that the operation is performed.

### 3.3 Suggestions for Compiler Improvements

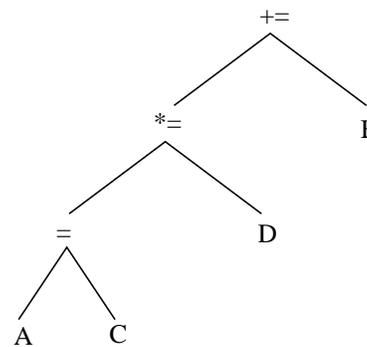
Consider again the expression analyzed earlier:

$$A = B + C * D;$$

This expression can be optimized by operator substitution. A good optimizing compiler could replace the original parse tree



with the parse tree



The obvious advantage is that no temporaries are used. A less obvious but equally important advantage is that no constructors or destructors are called between operations. Because no constructor/destructor calls separate the operations, a compiler that can inline loops can also chain the operations together. This implies performance near the theoretical maximum.

However, a number of assumptions are made when transforming the parse tree in this manner. These assumptions are:

- 1) The operations  $A=A*D$  and  $A*=D$  are equivalent.
- 2) The operations  $A=A+B$  and  $A+=B$  are equivalent.
- 3) The  $+$  operator commutes.

Note that these assumptions are true of both real numbers and of the Matrix class we described earlier. However, while a C++ compiler is allowed to make these assumptions about real numbers, no construct in the C++ language permits the compiler to make such assumptions about a user-defined class. An extremely intelligent compiler might be able to decide whether such assumptions are safe from the definition of the class and its inline operators; but such compilers are years away if they can be written at all. We feel that it would be advantageous to invent new C++ language constructs to inform compilers of which such common assumptions apply to the operators of a given class. For the present, such constructs might take the form of new `#pragma` directives.

#### 4. Conclusion

In the above discussion, it has been shown that C++ can perform efficiently as a scientific programming language. In order for C++ to be efficient, C++ library programmers will have to be aware of the cost of creating temporaries and utilize any vendor specific optimizations that may exist. By using C++ as the programming language, a code project can have a **single** user code that links with specialized libraries which seek to achieve the peak performance of a particular vendor's architecture. Lastly, we believe that with the features of C++, scientific programs can be developed in less time with fewer defects than with any other language.

#### 5. References

1. RHALE++, Computational Physics Research and Development Division, Sandia National Laboratories, Albuquerque, NM., private communication.
2. ZEPHR++, Computational Mechanics and Visualization Division, Sandia National Laboratories, Albuquerque, NM., private communication.
3. PCTH, Computational Physics Research and Development Division, Sandia National Laboratories, Albuquerque, NM., private communication.
4. ACIS, Spatial Technology Inc., Boulder, CO.
5. Ian Angus, Northrup Research and Technology Center, Palos Verdes Peninsula, CA., private communication.
6. T. J. Ross, L.R. Wagner, and G.F. Luger, "Object Oriented Programming in C++ for Scientific Codes," ASCE 8th Conference on Computing, June 10-12, 1992.
7. Dan Quilan, Department of Applied Mathematics, University of Colorado at Denver, CO., private communication.
8. D. Forslund, C. Wingate, P. Ford, S. Junkins, and S. Pope, "A Distributed Plasma Particle Simulation Code in C++," ASCE 8th Conference on Computing, June 10-12, 1992.