

## Experiences in Using C++ to Develop a Next Generation Strong Shock Wave Physics Code

James S. Peery<sup>1</sup> and Kent G. Budge<sup>1</sup>

### Abstract

The goals and time constraints of developing the next generation shock code, RHALE++, for the Computational Dynamics and Adaptive Structures Department at Sandia National Laboratories have forced the development team to closely examine their program development environment. After a thorough investigation of possible programming languages, the development team has switched from a FORTRAN programming environment to C++. This decision is based on the flexibility, strong type checking, and object-oriented features of the C++ programming language.

RHALE++ is a three dimensional, multi-material, arbitrary Lagrangian Eulerian hydrocode. Currently, RHALE++ is being developed for von Neumann, vector, and MIMD/SIMD computer architectures. Using the object oriented features of C++ facilitates development on these different computer architectures since architecture dependences, such as inter processor communication, can be hidden in base classes. However, the object oriented features of the language can create significant losses in efficiency and memory utilization. Techniques, such as reference counting, have been developed to address efficiency problems that are inherent in the language. Presently, there has been very little efficiency loss realized on SUN scalar and nCUBE massively parallel computers; however, although some vectorization has been accomplished on CRAY systems, significant efficiency losses exist. This paper presents the current status of using C++ as the development language for RHALE++ and the efficiency that has been realized on SUN, CRAY, and nCUBE systems.

### Introduction

Large computational physics codes are increasing in complexity as customers de-

---

<sup>1</sup>Senior Member of Technical Staff, Computational Physics Research and Development Division, Sandia National Laboratories, Albuquerque, New Mexico,

mand 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.

C++ is growing in popularity as a scientific programming language. At Sandia National Laboratories, the next generation shock wave physics code, RHALE++ (Peery, Budge, Robinson, and Witney, 1991) is being developed in C++. In addition, PCTH (Robinson, et. al., 1992), the parallel version of the 3-D Eulerian shock wave physics code, CTH, is being written in C++ and will be portable to SIMD, MIMD, and SDMD architectures by modifying lower level libraries. Moreover, Sandia development groups are using the third party product ACIS<sup>®</sup> (ACIS, 1991) 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 (Angus, 1992), University of New Mexico (Ross, Wagner and Luger, 1992), University of Colorado (Quinlan, 1991), and Los Alamos (Forslund, Wingate, Ford, Junkins, and Pope, 1992).

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.

#### *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 CFRONT 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.

### 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

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

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.

Programs that model physics typically require many types of fields: scalar, vector, and tensor fields (Budge, 1991). 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));
    VectorField InForce = Div(Rotated_Stress,
        CurCoor);

    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.

### 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 three unnecessary assignments and two temporary matrices (Peery, Budge, Robinson, and Witney, 1991). 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.

### *Reference Counting*

Reference counting (Coplien, 1992) basically eliminates the creation of objects in copy constructors and overloaded = operator. Reference counting can be imple-

mented in a class by adding an integer pointer to the private data of a class. With reference counting, unnecessary assignment operations are eliminated but only one temporary is eliminated. 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;
```

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

### *Container Classes*

For some computer architectures such as the CRAY YMP, dynamic memory allocation can be very time consuming. One method of eliminating inefficiencies associated with allocating and deallocating memory is to develop a class that manages its own data. Classes that manage themselves are called container classes (Coplien 1992). 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++ and run on a Sun SparcStation II, single processor CRAY YMP, and a single processor nCUBE. 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 on the mangled code that AT&T CFRONT produces. The nCUBE C++ executables were generated with the GNU g++ compiler. The performances of the three different codes on the different architectures are given in Table 1. Each code was compiled with the most aggressive compiler optimizations available. Different timings are given for C++ with reference counting (rc) and C++ with reference counting and container classes (rc and mm).

Table 1: CPU time (seconds) for matrix test case (rank 100, 10,000 iterations)

Language	SUN	CRAY (MFlops)	nCUBE (Vendor Libraries)
FORTRAN	110.1	0.71 (140)	185.8 (no)
C	108.4	0.77 (130)	150.8 (no)
C++ (default)	328.5	2.43 (41)	458.9 (yes)
C++ (rc)	131.9	1.06 (94)	82.8 (yes)
C++ (rc and mm)	130.4	0.83 (121)	81.4 (yes)

Table 1 shows that C++ coding performed well against FORTRAN and C coding under the conditions that one develops classes that use reference counting and perform their own memory management. However, the default C++ timings demonstrate the extent of inefficiency that can result from basic C++ programming. In contrast, the C++ coding generated for the nCUBE used vendor supported library functions to perform the “+” and “=” operations on the arrays which produced 2.3 times faster coding than FORTRAN. Although this is a simple code segment, we feel that this is a fair comparison because the change was made only in the matrix library and thus would be inherited in every place that a more complex code used the overloaded function. By developing code in C++, optimization made in a class function 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.

Notice in Table 1 that the FORTRAN coding achieved almost peak performance for a single functional unit on a CRAY Y-MP (166 MFlops). Because the CRAY Y-MP has two functional units, with chaining the peak performance is 333 MFlops. 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.

### 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 an optimized parse tree as shown in Figure 1. 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.

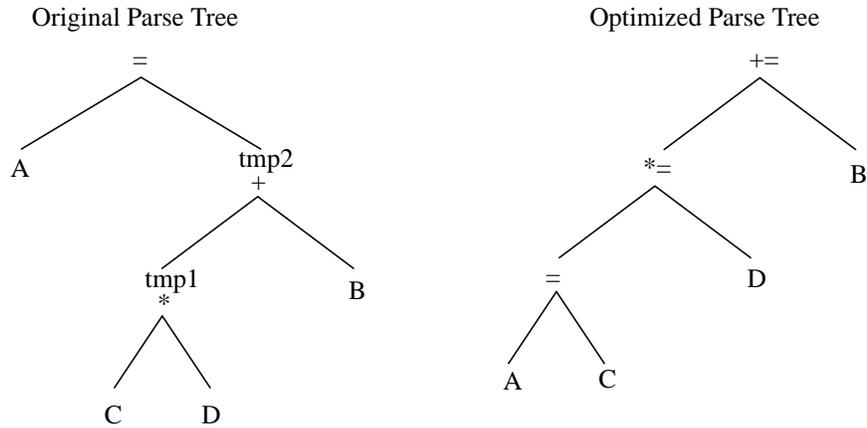


Figure 1. Example of Parse Tree Optimization

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.

## 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.

## Acknowledgments

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

## References

ACIS, Spatial Technology Inc., Boulder, CO., 1991.

Angus, I.G., "Parallelism, Object Oriented Programming Methods, Portable Software and C++," ASCE 8th Conference on Computing, June 10-12, 1992.

Budge, K.G., "PHYSLIB, A C++ Tensor Class Library," SAND91-1752, May, 1991.

Coplien, J. O., Advanced C++, Addison Wesley, Reading, Massachusetts, 1992

Forslund, D., Wingate, C., Ford, P., Junkins, S., and Pope, S., "A Distributed Plasma Particle Simulation Code in C++," ASCE 8th Conference on Computing, June 10-12, 1992.

Peery, J. S., Budge, K. G., Robinson, A. C., and Witney, "Using C++ As A Scientific Programming Language," CRAY User's Group Conference, Santa Fe, NM, Sept. 23-27, 1991,.

Dan Quilan, Department of Applied Mathematics, University of Colorado at Denver, CO., private communication, 1991.

Robinson, et. al., "Massively Parallel Computing, C++ and Hydrocode Algorithms," ASCE 8th Conference on Computing, June 10-12, 1992.

Ross, T. J., Wagner, L.R., and Luger, G.F., "Object Oriented Programming in C++ for Scientific Codes," ASCE 8th Conference on Computing, June 10-12, 1992.

# Experiences in Using C++ to Develop a Next Generation Strong Shock Wave Physics Code

James S. Peery<sup>1</sup> and Kent G. Budge<sup>1</sup>

## KEYWORDS:

C++  
computational  
physics  
efficiency  
shock  
wave  
object  
oriented  
programming  
parallel  
vector