



Fermi National Accelerator Laboratory

FERMILAB-Conf-91/159

C++ Objects for Beam Physics*

L. Michelotti
Fermi National Accelerator Laboratory
P.O. Box 500
Batavia, Illinois 60510

June 1991

* Presented at the *14th Biennial IEEE Particle Accelerator Conference*, May 6-9, 1991, San Francisco, CA.



Operated by Universities Research Association Inc. under contract with the United States Department of Energy

C++ Objects for Beam Physics

LEO MICHELOTTI

Fermilab*, P.O.Box 500, Batavia, IL 60510

1 Introduction.

Let us begin by admitting that the movement of physicists from FORTRAN to C++ has been less than a groundswell. At the same time, let me restate my opinion that scientific programming can be facilitated by an extensible language – such as C++ , Objective-C, or ADA – one in which we can define, easily and naturally, new variable types that behave, in all respects, like *fully functional variables of the language*. C++ is tailored to suit programmers' needs by creating “classes,” which specify (a) structures of data, (b) the functions and operators which act upon them, and (c) rules¹ for creating and annihilating them. Among the advantages of working within such a language are:

- **user-friendliness** Type checking, operator and function overloading, and data hiding help in building user-friendly C++ classes that behave as close to “expected” as possible. For example, operator overloading means that arithmetic on algebraic classes can be implemented with the usual tokens: +, -, *, and /. Function overloading means a statement like “ $y = \cos(x)$ ” will work regardless of whether x and y are of type `double`, `complex`, `matrix`, `quaternion`, or any other type for which the statement makes sense. If data conversions are appropriate, such as might arise in mixed mode arithmetic, the class constructors can handle this detail themselves. Other routine bookkeeping tasks can be imbedded within class implementations, freeing the application program(mer) from them.

- **inheritance and extensibility** The class `DA` , which is discussed below, is an implementation of differential algebra in C++ . Having built it, one can easily go on to define other classes, such as `DAMatrix` – matrices whose elements are `DA` variables – with arithmetic operators overloaded in the obvious way. Should it be useful to do so, we could also have `DAquaternion` or `DAcomplex` variables, and one could go on to develop toolkits for `rational`, `group`, or any mathematical object that might be useful.

*Operated by the Universities Research Association, Inc. under contract with the U.S. Department of Energy.

¹The constructor and destructor functions which bring variables into and out of scope.

- **language support** The advantages of working within a supported language should not be undervalued. Once defined, classes have *the full functionality of any other variable type within the language*. Suppose that a class “`zlorfik`” has been defined. This functionality means that an applications programmer can: (a) declare `zlorfik` variables just as easily as `double`, `int`, `char`, or any other type of variable, (b) write functions which return a value of type `zlorfik`, (c) declare `zlorfik` aggregates – multi-dimensional arrays, lists, trees, or whatever – and (d) apply class operators not only on explicitly declared `zlorfik` variables but also on expressions which evaluate to type `zlorfik`. One gets all this for free – not even the class designer has to sweat the details – by working within a language *designed to be extensible*.

But enough missionary work. We shall describe below a few C++ classes and applications that have been written recently at Fermilab for problems in accelerator physics.²

2 MXYZPTLK

In a paper for the previous IEEE PAC I described a C++ class, `nstd`, which directly and very straightforwardly implemented automatic differentiation. Although it could (and did) do simple calculations, it was designed more for pedagogy than practical work. MXYZPTLK was designed to correct serious inadequacies of `nstd`. In it are defined two classes, `DA` and `DAVector` , which implement the “prolonged numbers” of `nstd` as dynamically allocated (and deallocated) doubly linked lists — that is, they are derived from the container class `dlist` — with attributes defined at runtime.

The `DA` and `DAVector` classes possess a unary operator, `.D` and a function `.derivative`, which correspond to performing and evaluating a derivative, as shown below.

²The new limit of three 8.5 × 11 pages has made it impossible to provide a bibliography, so there will be no detailed citations of connected works. However, the major players in the games these tools are meant for are: Alex Dragt, Etienne Forest, Martin Berz, Filippo Neri, Johannes van Zeijts, Ron Ruth, Robert Warnock, Yiton Yan and so forth.

```

DA      u, v;
double  x;
int     m[] = { 2, 1 };
...
v = u.D( m );           // Line A
x = u.derivative( m ); // Line B

```

Line A corresponds to a functional equation, $v = \partial^3 u / \partial x_0^2 \partial x_1$, while Line B merely loads the value of this derivative into the variable `x`. It is the operator `.D` that makes DA objects into a “differential algebra,” in the sense of Berz.

In addition to the unary operator `.D : DA → DA` there is a binary operator `^ : DA × DA → DA` which implements Poisson brackets. For example, the program fragment below is used to evaluate the Poisson bracket of the two expressions,

$$a = x_1 x_2^2 p_1 p_2^3, \quad b = \sin(x_1 p_2^2 x_2^3).$$

```

DA  x1, x2, p1, p2, a, b, pb;
...
a  = x1 * (x2*x2) * p1 * (p2*p2*p2);
b  = sin( x1 * (p2*p2) * (x2*x2*x2) );
pb = a^b;
cout << pb.standardPart();

```

The last line prints the value of the bracket to the screen. Because instances of C++ classes are fully functional variables, this could have been done in one step:

```

cout << ( ( x1 * (x2*x2) * p1 * (p2*p2*p2) )
          ^
          ( sin( x1 * (p2*p2) * (x2*x2*x2) ) )
        ).standardPart();

```

which applies `.standardPart` to the *expression* obtained by taking the bracket of the two *expressions* formerly loaded into `a` and `b`. In this same vein, the Jacobi identity can be tested among three expressions, `a`, `b`, and `c` with the line,

```
( a^(b^c) ) + ( b^(c^a) ) + ( c^(a^b) ) .peekAt();
```

which prints all non-zero members of the expression to the screen. Indeed, the arguments to the bracket operator (or any DA function) may also include *functions which return a value of type DA*.

Each DA variable keeps track of its own attributes, such as accuracy and reference point. Because the order of derivatives kept in the list is necessarily truncated, the DA value resulting from an invocation of `.D` or the Poisson bracket operation is not as accurate as the arguments that went into it (its highest derivatives are missing). If this variable is used later in calculations, the results are also less accurate. Such information is carried along and upgraded automatically across computations. If the application program tries to differentiate a variable too many times, to access derivatives which are not accurate, or to multiply two DA variables that have different reference points, an error message will be printed.

Concatenation is implemented in MXYZPTLK via a binary operator, `*`, applied to variables of type `DAVector`. Although it superficially acts like an array of DA variables, in fact a `DAVector` has extra structure.

The first version of MXYZPTLK was released in January, 1990. Source code can be obtained upon request, and a User’s Guide has been written. It is worth mentioning that MXYZPTLK implements automatic differentiation by simple forward-mode algorithms and would probably not be suitable for large-scale problems requiring hundreds of coordinates. These may be handled better by reverse-mode methods incorporated into ADOL-C, for example.

3 beamline

The very name of the class `beamline` suggests what it is. `beamline` is derived from two parents: `dlist`, a container class which implements a `beamline` variable as a doubly linked list, and `bmlnElmnt`, a base class that contains information common to all beamline elements, such as geometry, pointers to circuits, or conversion factors (e.g., amperes to Tesla). That `beamline` is derived from `bmlnElmnt` makes it easy to insert one `beamline` variable into another.

A C++ program fragment that builds a lattice consisting of five identical FODO cells may look as follows.

```

double  length, focalLength;
...
drift   0 ( length );
thinQuad F ( focalLength );
thinQuad D ( - focalLength ); // Line 5

beamline A ( &F );
A.append ( &O );
A.append ( &D );
A.append ( &O ); // Line 10

beamline B;
for( int i = 0; i < 5; i++ ) B.append( &A );

```

Done in this way, any subsequent adjustment of the focal length of the F quad will take place simultaneously in all “five” cells of the lattice. An alternative to building the `beamline` variable element by element is to declare it with a string argument, which is interpreted as a lattice file. The statements,

```

beamline Tevatron ( 'lowBeta.synch' );
beamline mainInjector ( 'mi_17.flat' );

```

declare two `beamline` variables, the first defined in a SYNCH file, the second in a FLAT format file.

The `beamline` class interface contains the critical lines,

```

virtual void propagate( double* );
virtual void propagate( DAVector& );

```

which establish that every specific beamline element must contain two `.propagate` member functions. The first, which accepts an array of (six) real variables as its argument, does straightforward, element-by-element tracking through the beamline; the second, which accepts a `DAVector` as its argument, constructs the polynomial map corresponding to concatenating the maps of its elements.

The `beamline` class itself has `.propagate` member functions that do the same. One of the extraordinarily useful features of the `virtual` statement is that we can do this sort of thing so easily. The entire source code for the `beamline::propagate` function consists of two declarations and one executable line.

```
void beamline::propagate( DAVector& x ) {
dlist_iterator getNext ( *(dlist*) this );
bmlnElmnt* p;
while ( p = (bmlnElmnt*) getNext() )
    p -> propagate( x );
}
```

The beamline is being told to go element by element and propagate `x` through each one. There is no sequence of decisions to determine what to do based on type. Each type of variable knows itself what it is supposed to do.

And how does it know? One feature of the `bmlnElmnt` class is that *all the physics is isolated and contained in a collection of .physics files*. For example, the `.propagate` implementation for a thin quadrupole element contains the line,

```
{ #include "thinQuad.physics" }
```

which file is to contain all the physics associated with passage through a thin quadrupole, the rest being boilerplate and logistics. This file may contain only the lines

```
upr = upr - ( u / f );
vpr = vpr + ( v / f );
```

where `f` is a part of the private data of a `thinQuad` beamline element, representing the focal length of the quadrupole. Now, someone may very well object to using this, as it makes no mention of longitudinal momentum. He has the option of using an alternative file from the `thinQuad.physics` library, say,

```
upr = upr - ( u / ( f*( 1.0 + wpr ) ));
vpr = vpr + ( v / ( f*( 1.0 + wpr ) ));
```

where `wpr` is to be interpreted as $\delta p/p$. In addition, he can tinker with the files on his own. Suppose a user wants to do something unforeseen, such as call on a new symplectic numerical integrator to go very carefully through a thick element. He has the freedom to create his own `.physics` file, and the changes would be completely transparent to any application software using `beamline`. The brackets surrounding the `#include` statement assure that *variables declared or dynamically created by such tinkering can never interfere with variables of the same name in other parts of*

the class source code. The C++ scoping rules will prevent it – another feature one has for free when working within the language. Our hypothetical tinkerer need never look at boilerplate and logistics.

A generic type of `bmlnElmnt` can exist in different flavors. For example `quad` is a derived `bmlnElmnt`, as are `thinQuad`, `DAQuad`, and `DAtinQuad`. The implication of `thinQuad` is obvious; the other two refer to a quadrupole whose properties – length and strength – are themselves `DA` variables, enabling them to be used as control coordinates in optimization calculations or to appear in polynomial expansions.

The above was written as though the design were already implemented. Some is; more is not. The first working version of `beamline` should be ready in Fall, 1991.

4 AESOP and canvas

AESOP³, was introduced and demonstrated at the last IEEE PAC, so we shall not dwell on it here, as space is getting short. Suffice it to say that AESOP is a Phigs-based, prototype graphics shell, programmed in C++, for implementing “exploratory orbit analysis.” Its objective is easy, interactive exploration of four-dimensional phase space maps. AESOP’s “four-dimensional cursor,” implemented late last year, greatly facilitates the finding and tracking of four-dimensional separatrices through bifurcations. (See *Resonance Seeding of Stability Boundaries in Two and Four Dimensions*, this Proceedings.) A script (VAX/VMS DCL command file) is provided for linking AESOP with any four-dimensional mapping routine, which may be written in C++, C, or FORTRAN. It has been used at Fermilab to explore the offset beam-beam interaction in the Tevatron, space charge in the booster, and in the Main Ring.

AESOP was written originally for the Evans and Sutherland PS390, but I plan to port it to the Sun environment “soon.” The PS390 is a powerful, sophisticated graphics engine which is largely underutilized. The reason for this lack of enthusiasm is that no scientist of sound mind would read, much less assimilate, its seven-volume set of manuals. To respond to this problem of making it easier to view two- three-, and four-dimensional data on the PS390, a C++ class, `canvas`, was built. By simply declaring `canvas` variables application programs are provided with objects that accept (scatterplot, wireframe, or vector-field) data and display them automatically. The “real-time” transformation capabilities of the PS390 are activated in one step by “connecting” its external devices, the dials and the puck, to the desired `canvas`. A `rasterCanvas` class is also available for scanning two-dimensional regions (ala Mandelbrot, for example).

³ Analysis and Exploration of Simulated Orbits in Phasespace, or some such thing.