# Reflections on a Decade of Object-Oriented Programming in Accelerator Physics

Leo Michelotti

*Fermi National Accelerator Laboratory*[†]
*P. O. Box 500, Batavia, IL 60510, USA*

Let me begin by stating the obvious: there is neither enough space in a three page paper nor was there enough time in a fifteen minute talk to do justice to the title printed above. Before too many more years pass, someone should undertake a serious psycho-sociological study of scientific C++ programming in the 20$^{\text{th}}$ century. There is substantial paydirt to be sifted, enough to generate a score of Ph.D. theses on human behavior and a half dozen volumes on the philosophy of science. Invaluable anecdotal information can be mined on the processes of software acceptance by physicists. If such books are ever written, I would readily contribute a few chapters on sundry proven prescriptions for failure. In anticipation, a few personal reminiscences are scrawled here.

## MOTIVATED TO OOP

Three unrelated circumstances led me, in 1988, to search feverishly for new ways of programming. The first was my use of a remarkable graphics engine, the Evans & Sutherland PS390, to display projections of four-dimensional orbits. The PS390 was programmed using a Phigs-like graphics protocol. Even though everything was done with FORTRAN subroutine calls, the underlying graphics model was object oriented. One constructed a data processing net *by combining simple objects*: translators, rotators, scalars, and so forth. Some of these objects could be linked to external peripherals, such as knobs, which enabled real-time manipulation of both data and display.

At the same time, a minor revolution was taking place in the way that (a few) accelerator physicists were doing calculations. Rapidly increasing computing power, coupled with some clever ideas, encouraged the introduction of old mathematical objects – Lie algebras, exponential maps, differential algebras – into new problem domains. A desire to program these algebraic manipulations within a natural syntax moved me to search for a language that permitted operator overloading.

The third circumstance was a strong personal aversion to the “monolithic code paradigm,” in which accelerator physics calculations are carried out by submitting an input deck to a single program expected to do everything. As an alternative, I thought it would be useful to have a collection of *simple objects that could be combined to do complicated things*, as was done in Phigs. At the same time, I didn’t believe that physicists should be creating their own programming languages, which some were doing. If a *real* language could be used both for describing beamlines and doing calculations, then there would be no need of specially formatted input decks or their parsers; the language itself would be the parser. Physicists could then write their own applications using this toolkit, rather than relying on the monolithic codes.

Such was the goal, anyway. A few requirements for its achievement were: (a) clean separation of physics from geometry and/or hardware considerations; (b) local, not global, coordinates to be used throughout; (c) default “propagators” – objects containing the physics – supplied for each type of beamline element, but specialized ones could be plugged in if desired; (d) automated algebraic algorithms to be implemented expressively; (e) an ordered hierarchy of libraries to make calculator objects work seamlessly with beamline objects; (f) identical coding to be used for tracking and analysis.

It was in this frame of mind that I first ran across *The C++ Programming Language*, by Bjarne Stroustrup: a small paperback, only 15 mm in width, recommended to me by my colleague Michael Allen. What followed is summarized briefly in the next section.

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

## A TELEGRAPHIC PERSONAL TIMELINE

**Phase 1, 1988-1992: Beginnings.** Terrible compilers and no debuggers. Learning a new approach to programming in surprisingly uncooperative sociological and technical environments. Rapidly evolving "vision." Continually rewriting software: version after version. Spectacularly unsuccessful attempts at proselytizing. Redeeming feature: C++ was a simple, powerful, inexpensive language that allowed me to program the way I wanted. Wrote first versions of BEAMLINE (modeling accelerator components) and MXYZPTLK (algebraic classes) libraries by 1989. (Answer to the one and only MXYZPTLK FAQ: it's pronounced ZIE-tul.)

**Phase 2.1, 1992-1993: Communication.** First contacts with like-minded people. Learn quickly that writing robust OOP/C++ classes useful to others is *very* challenging. Begin to profess: "Using good classes is easy; writing them is hard."

**Phase 2.2, February 16-19, 1994: The Workshop.** I convene a "Workshop on C++ Classes for Design, Analysis, Modelling, and Control of Accelerators and Detectors." Approximately 30 people show up to share ideas. Many exchanges; few points of agreement. Presentation of my BEAMLINE/MXYZPTLK libraries leads to the one almost unanimous consensus that I don't know what I'm doing. (After the workshop, Christoph Iselin returns to CERN and delivers a seminar suggesting that writing MAD 9 be replaced with developing a set of class libraries. This idea is not received enthusiastically.)

**Phase 2.3, 1995-1997: Collaboration.** At the 1994 Workshop, Paul Kunz suggested that accelerator physicists devise a common interface, a set of header files that "everyone" could agree on. Shortly afterward, a collaboration, called CLASSIC, is formed, loosely under John Irwin's leadership, to do that. Several collaboration meetings are held; I attend one and kibitz the others. Christoph Iselin, arguably CLASSIC's most prolific member, writes MAD 9 in C++ . Others go in various directions and write their own class libraries. The collaboration no longer exists in any meaningful sense.

**Phase 3, 1996-1999: Withdrawal.** Concluding that an international software collaboration of physicists is like an orchestra of manic musicians, all playing their own compositions, I return to doing calculations in support of local projects. Joel Butler hides me in the Computing Division (CD) for awhile, and I improve the BEAMLINE/MXYZPTLK libraries by adding new classes, new functionality, and making everything more user-friendly. Afterward, an ambitious online modelling (OLM/OAM) effort is begun: connecting accelerator models with control consoles. At a critical juncture, 2/4 of the OLM/OAM team leave Fermilab. Elliott McCrory and Jean-Francois Ostiguy program heroically, salvaging what can be, but the project's long term viability is doomed.

**Phase 4, 1996-1998: FPCLTF (pronounced ZOOM).** CDF and D0 decide to rewrite software in C++ . Informally attending brain-storming sessions at D0, I eventually join the FPCLTF, a laboratory supported, organized, C++ support group, and learn more good programming techniques. Unfortunately, CD chooses to support only detector-related HEP activities.

**Phase 5, 1999-present: To oblivion and beyond.** OLM/OAM disintegrates quietly from lack of support. I join FNAL's NLC team, upgrading the BEAMLINE/MXYZPTLK libraries to study linacs. We now have excellent compilers and debuggers, but C++ is no longer a simple language. Controls Department, never having accepted C++ , is moving toward JAVA.

## PROGRAMMING ISSUES

Using C++ so early in its development meant running headlong into problems and anticipating solutions that did not yet (or still do not) exist. Some of them are presented here.

**Run time type identification (RTTI).** From the beginning, long before RTTI was part of the language, it was essential to be able to clone an object given only its address, without knowing its type. To accomplish this, classes inherited from an abstract base class which contained a pure virtual function, *.Clone()*: defined as returning a pointer to an instance of the base class, implemented in the derived classes by returning a pointer to a new copy of the object. This left responsibility to the invoking module for (a) recasting properly and (b) deleting the object. (Such responsibilities are considered dangerous and offend purists.) Querying an object's type was done similarly: the base class contained a pure virtual function *.Type()* which, as implemented in the derived classes, returned a string identifying the class.

**Ownership.** An instance of `class beamline` contains a set of elements – quadrupoles, dipoles, cavities, and so forth – any one of which may be other beamlines. I made an early, but probably correct, decision that a `beamline` should not own its elements. (This is opposed to the STL philosophy, which came later.) Thus, a `beamline` does not have responsibility for its objects, and their handles are available for outside influence. (The same object can even appear in several positions.)

**Garbage Collection.** Large containers of volatile objects are created and destroyed on the fly, and copy constructors reproduce data unnecessarily. My earliest attempt to deal with these problems involved inheriting from a base class, `Junk`, which possessed a virtual destructor. All dynamic, volatile objects were derived from `Junk`, and their pointers were stored in an `Attic` object. When the number of `Junk` objects reached some critical value, a percentage of the oldest were deleted.

This was, of course, a foolishly risky scheme, which I abandoned after learning the "envelope-letter idiom" from Coplien's *Advanced C++ Programming Styles and Idioms.* This fundamental procedure was an eye-opening introduction to the values of reference counting. Nothing would be deleted until it was no longer needed; garbage collection occurred on the fly; data existed only in one place; no large objects need ever be copied.

A serious problem remains: the inefficiency of dynamically creating and destroying large numbers of objects. To get around that, we pre-create pools of objects to draw from (and replace). However, this works well only for objects of well-defined size.

**Persistence.** Persistence is important! The C++ language does not provide it; perhaps it never will. Instances of `class beamline` and their elements must persist, available for use by multiple applications. We tried various pickling and swizzling techniques: early and simple solutions utilized streaming and virtual "writeTo" functions. However, essential tasks of providing object pointers and function pointers were not handled correctly. Our most recent solution uses Joseph Hesse's `EZSave` and seems to be adequate for now. In this context, a "Factory" class is almost indispensable. (Presumably, the best solution is an object oriented data base.)

**Ferrets and Visitors.** At first, adding functionality to `class beamline` meant continually modifying basic header files and source code. However, many calculations proceeed in a sequential manner, processing one beamline element at a time. An early CLASSIC idea was the "Ferret" or "Algorithm," a base class for objects that would propagate through beamlines and do specific calculations by changing state as they traversed the elements. This was superseded by the visitor pattern, which we adopted enthusiastically.

**Barnacles.** Different `calculator` objects (e.g., visitors) may be able to use information generated by earlier calculations. Without invoking centralized organization, how can a later `calculator` know whether the earlier calculation has been done and, if so, where to find the information? `Barnacle` is a base class for a "Post-It note," if you will, that could be attached to individual beamline elements. A calculator would attach its own information, in an object derived from `Barnacle`. The information can be retrieved later via a keyword search. (This was something akin to an STL multimap, except that the data could be anything whatsoever and need not be of uniform type across all the `Barnacles`.)

**Functors.** The goal of separating physics from other considerations, while allowing for dynamic specialization, was achieved with the exceedingly useful concept of functors. Default "propagator" functors are provided for every beamline element class. Specialized ones can be written by users – e.g., to employ higher order symplectic integrators. Propagator states can be adjusted by control objects; they can be bound to elements dynamically. Functors satisfied all the required criteria.

**Dynamic Polymorphism.** At times, it is convenient for a variable to "change its type" in the midst of a program – e.g., from a `double` to a `polynomial` – without changing its name or address. This can be mimicked using the envelope-letter idiom, wherein an object is only a container for a pointer to data. Double dispatch provides a mechanism for handling operator overloading in algebraic classes. There are several ways to do this; I chose to maintain "doubly indexed function tables."

**Sages.** People misuse software, sometimes by thinking incorrectly about physics (e.g., trying to calculate a closed orbit with RF cavities energized). Without resorting to the monolithic code paradigm, I tried to make calculations more foolproof by introducing `Sage`, a base class for objects that know how to use correctly lower level calculator and beamline classes to do a particular category of calculations . In fact, this comes dangerously close to writing monolithic code and fundamentally violates the original intent of the BEAMLINE/MXYZPTLK. The libraries were not supposed to be foolproof; they were meant to be used by those who understood at least the physics and mathematics of what they were doing. There should have been no need for a `Sage`, but human nature dictated it.

**Broadcasting; event handling.** It is frequently necessary to inform certain categories of objects about events that have occurred, such as modifications to a beamline. This information must be broadcasted to all existing objects possessing this "need to know." Again, here is a problem not solved by the language itself. It is the sort of problem that arises continually in graphics applications, and the solution can be found therein. Basically, a unary central message server must exist which maintains a registry of all objects responding to particular kinds of events. Objects register (deregister) with this server via their constructors (destructors). (Jean-Francois Ostiguy has become a local expert on this; I am indebted to him for these observations.)

## NON SEQUITUR CONCLUSION

There are a minimum of two distinct levels at which C++ can be used: object-based programming (OBP, using objects, which is easy) and object-oriented programming (OOP, creating object classes, which is hard). Physicists who have broken free of the monolithic code paradigm have a natural inclination to peer into nooks and crannies, leading to an inverted population of OOPers over OBPers. At one time this was unavoidable, because classes did not yet exist. That is no longer the case, but the natural inclination is hard to overcome.