PROGRAMMING LANGUAGES AT LHC

I. E. Zacharov
CERN, Geneva, Switzerland

Abstract

The evolution of the computer hardware and software requires a change in the programming philosophy of the LHC experiments. Fortran 90, the successor of FORTRAN 77 is a likely candidate to be used in the new computing environment. However, if the trend towards the object-oriented languages shows the practicality of this new approach, the C++ programming language is a better choice for the LHC experiments.

Presented at the ECFA Large Hadron Collider Workshop
4-9 October 1990, Aachen, Germany

*TEX: F90C.TEX.A on CERNVM e-mail: ZACHAROV@CERNVM
1 Introduction

With LHC a few years away, now is the time to (re)define the programming support CERN can offer for future experiments. The background for the rethinking process are the difficulties with the software experienced by the HEP community and the software crisis in general.

Therefore it is essential for the success of future LHC experiments to plan carefully their software development. CERN can bring out some recommendations in this area, and based on a general agreement it has to decide which libraries, data bases and packages it will offer to the physics community. Especially important is the decision on which programming languages CERN will support.

It is clear to the majority of the programmers that software written in the FORTRAN 77 programming language tends to be less reliable than what would be reasonably expected from another, strongly typed language. Therefore, despite the popularity FORTRAN 77 has among the physics community, it would be desirable to write the code in a programming language offering support for modern software engineering methods. An important criterion for a new language is the level of data abstraction it can offer to the programmer.

One attractive choice as a modern programming language seems to be Fortran 90, a successor to FORTRAN 77. It is now in its last stages of formal standardization. Another attractive possibility is the C++ programming language, which is already available on most platforms. This report lists arguments in favour of each of these programming languages, considering the features of both languages and the environment in which they will be used.

The ADA programming language is not discussed in this report. From the software engineering point of view Fortran 90 offers the same possibilities as ADA, avoiding most of its heavy constructs. On the other hand ADA does not have the class inheritance mechanism like of C++, therefore it is not an object oriented language either. At the moment there is no appealing reason to use ADA for the HEP programming. However, not everybody would reject ADA so readily. The discussion in this report can be viewed as a comparison of a powerful, computational, procedural language (like Fortran 90) with an efficient object oriented language (like C++).

This comparison of the programming languages is not a comparison of the semantics. It is rather the comparison of the underlying philosophy; a decision about the programming style leads to the choice of a programming language supporting it. Boundary conditions have to be taken into account, however, which might inhibit certain choices of programming languages.

The report is organized as follows. We consider the Computational Model and the Computational Environment of the 90's in the next chapter. Chapters 3 and 4 discuss respectively Fortran 90 and C++; chapter 5 expands on the concept of the inheritance.

2 The Computational Environment

A computer program is never self-sufficient: it needs interaction with the Operating System (OS), e.g. I/O. Moreover, it has to be aware of the underlying hardware architecture to some extent, as big run time savings can be achieved with programs exploiting hardware capabilities.

2.1 The Computational model

The preferred hardware organization of the 90's is a collection of loosely coupled processors (Workstations) with tape and file server(s) on the network. There might be a powerful central
processor or several special purpose processors coupled to this network as well. This model is imposed by the evolution of the hardware.

In such an environment the Communication and the Naming Service, Resource Allocation, Remote Procedure Calls (RPC) and the associated with it Location Broker (just to mention a few concepts) obtain crucial importance. Moreover, it is a multivendor environment in which standardization is the only way to achieve connectivity.

In fact, this Computational model is already a reality today, and it will be enhanced in the future:

- with the introduction of faster networks (FDDI, HIPPI, etc.)
- yet more powerful workstations (20 Mips workstations are available today)
- an adequate distributed programming support (especially improving the cpu utilization in a loosely coupled environment)

It is estimated that on the timescale of two years (in 1992/1993) this Computational model will be dominant in the HEP environment.

2.1.1 Note on the Distributed Processing

Generally speaking there are two ways to use a system of loosely coupled processors:

- As a pool of processors for batch work; a well established method.
- Execution of tasks on several processors in parallel on behalf of a single program.

The HEP programming can profit from the well known method of automatic program parallelization using the explicit data parallelism (assuming events are independent), that serves both methods of processor utilization. The parallel task execution on behalf of a single program can be programmed using RPC.

It is important to emphasize that the RPC or any other remote task execution method needs an explicit interface definition, so that stubs can be generated automatically for the client and the server. Therefore this Computational Model demands a strongly typed programming language with explicit interface definitions.

2.2 The Operating System

The Operating System of the 90's is UNIX. Although introduced initially on small machines, nowadays UNIX is supported on the most powerful mainframes (Amdal, nCube, just to mention a few; IBM is moving in that direction).

There is a considerable number of flavours of the UNIX Operating System on the market. It should be emphasized, however, that the differences between the UNIX flavours are more important for system programmers than for "normal" users. It is much more easy to move an application between different UNIX platforms than from VMS to MVS for example. Usually vendors support something called AT&T sys5.3 with BSD4.3 enchancements.

From the many flavours of UNIX two main streams are emerging right now: the "Open Software Foundation" (OSF) type of UNIX (supported by DEC, IBM, HP, etc., about 250 vendors) and the "UNIX International" (UI) type (most notably AT&T, Sun and Amdal, about 200 vendors). Many vendors are members of both, OSF and UI and will support both flavours in their products.
Programs running in the UNIX environment request OS services via a call to a standard library, passing parameters packaged in structures. This model requires the programs to be written in a programming language which supports derived data types. The programming language in the UNIX environment is C. Fortran (77 and 90) is intrinsically incompatible with UNIX and demands a special binding standard to be able to access OS services. For the UNIX interface to the FORTRAN 77 programming language the standardizing committee will go to ballot this year (1990) (the IEEE 1003.9 FORTRAN bindings: a library of fortran callable routines).

2.3 The Programming Base

There is a huge investment in software written in the FORTRAN 77 programming language. Therefore, whatever new programming language is adopted by the HEP community, it should allow a smooth transition from the present situation. Also the investment in training and experience is important.

However, it is not entirely clear that the investment in software is as big as it seems to be. New experiments will develop software (almost) from scratch. The memory management systems as we know them today (e.g. ZEBRA, etc.) might not be needed in their present form and will be partially replaced by the programming language constructs.

The balance between rewriting and reusing existing code is not found yet, and it will depend very much on the choice of the programming language.

2.4 Software Engineering

There will be a huge amount of data to be processed in the LHC experiments. Therefore, the software must be designed for efficiency. Yet, it is of enormous importance to increase the reliability of programs and to decrease the development time. The reusability of code is an important issue in this context.

Instrumental in this approach is the employment of modern programming methods. We do not claim that the drawing of Data Flow Diagrams is a panacea. What is important is the software development policy enforced by the compiler, that would automatically block the appearance of certain bugs in the system. For example, if the data is hidden, you are sure nobody except a well defined routine can modify them. If there is a subroutine interface, nobody can pass an invalid parameter. Data which describes itself can signal an inconsistent usage. If these (and some other) concepts are part of the programming language (i.e. they are enforced by the compiler), more reliable programs will be produced in shorter time.

2.5 Summary

To summarize, the computing environment of the 90’s demands a programming language with strong typing, explicit call interfaces, derived data structures, dynamic memory management and efficient computational support. As a matter of policy the concepts of loose module coupling, data hiding, self describing data should be enforced by the compiler.

3 Fortran 90

There are a few excellent books on the Fortran 90 standard (see e.g.[1]). Without attempting to be complete we will try to give a flavour of programming possibilities in Fortran 90. Another example is constructed by M. Metcalf in a recent article, see[2].
3.1 Module Coupling

It is well known that the tight coupling between modules introduced by fortran common blocks is a source of bugs in programs. In Fortran 90 the common block construct of FORTRAN 77 can be replaced by the MODULE declaration. We illustrate it with an example of the Standard Common Block proposed for the Monte Carlo (M.C.) program construction[3]. The most important F77 code is as follows:

\[
\begin{align*}
\text{PARAMETER (IEVMAX=2000, IEVSIZ=23, IEVDIM=IEVMAX*IEVSIZ)} \\
\text{COMMON /HEPEVE/ IEVRUN,IEVNUM,NEVTOT, particle_record, IHEP(IEVDIM)} \\
\text{EQUIVALENCE (IHEP(1),QHEP(1))}
\end{align*}
\]

Here, the particle_record is a collection of 23 variables, which can be filled by the M.C. programmer with the values of the generated particle properties (momentum, spin, etc.). Then, with the CALL HEPHIS(N,'S') they are transferred into the array IHEP at position N. Equivalently, if the array has been filled from a M.C. output, the CALL HEPHIS(N,'G') will set the particle record variables to the parameters of particle N. Potential bugs can enter the programs if users wish to access the IHEP array directly, without the HEPHIS routine. For example the IHEP/QHEP confusion is very common. The array boundaries are not protected either: outside HEPHIS the check NEVTOT.LE.IEVMAX cannot be enforced.

In Fortran 90 the construction might be as follows (note, that Fortran 90 is case insensitive; here we write the Fortran 90 keywords with capital letters for clarity):

\[
\begin{align*}
\text{MODULE hepeve} & \quad \text{! in line comments; names can be up to 31 characters} \\
\text{INTEGER ievrun, ievnum} & \quad \text{! the M.C. run and event number} \\
\text{INTEGER nevtot} & \quad \text{! number of particles generated so far} \\
\text{TYPE particle_record} & \quad \text{! user defined data type} \\
\text{INTEGER status, pdg, mother1, mother2, daughter1, \ldots (not all listed)} \\
\text{REAL px, py, pz, energy, mass, x_vertex, y_vertex, z_vertex, \ldots} \\
\text{END TYPE particle_record} \\
\text{INTEGER, PRIVATE, PARAMETER :: ievmax = 2000 ! not visible outside the module} \\
\text{TYPE(particle_record), ARRAY(ievmax) :: ihep} \\
\text{END MODULE hepeve}
\end{align*}
\]

This module (defined in a separate file) can be included in a user's program. Every routine which USEs a MODULE shares the public variables and the operators in it.

\[
\begin{align*}
\text{USE hepeve, ihep => myhep} & \quad \text{! optionally resolving potential name conflicts} \\
\text{IMPLICIT NONE} & \quad \text{! this useful setting is now a part of the standard} \\
\text{myhep(n)%pdg = 20} & \quad \text{! structure elements are accessed} \\
\text{myhep(n)%mass = \ldots} & \quad \text{! with the % qualifier} \\
\text{printf, myhep(n)%energy} & \quad \text{! such code removes the IHEP/QHEP confusion etc.}
\end{align*}
\]

An array boundary check can be enforced by making the ihep array PRIVATE and defining the HEPPAR interface:

\[
\begin{align*}
\text{MODULE hepeve} \\
\text{\ldots (up to the array declaration as before)} \\
\text{TYPE(particle_record), ARRAY(ievmax), PRIVATE, TARGET :: ihepx} & \quad \text{! private data} \\
\text{TYPE(particle_record), POINTER :: ihep => ihepx(1)}
\end{align*}
\]
INTERFACE hepar
  MODULE PROCEDURE hephis
END INTERFACE hepar
CONTAINS
  FUNCTION hephis(n) ! here is the subprogram body
    TYPE(particle_record), POINTER :: ihep
    INTEGER, INTENT(IN) :: n ! contains particle index
    IF(n <= 0 .OR. n >= ievmax) THEN
      IF(nevtot >= ievmax) CALL error ! no more room in the array
      nevtot = nevtot + 1 ! add a new particle in free space
      n = nevtot
    ENDIF
    hephis => ihepx(n) ! associate pointer with the n-th particle record
  END FUNCTION hephis
  SUBROUTINE error ! internal entry, not visible outside of the module
    PRINT*, 'no more room for additional particles'
    STOP ! could allocate some more space in reality
  END SUBROUTINE error
END MODULE hepeve

Now the user is forced to use the code myhep=>hephis(n) before the assignment.

For simplicity, we did not make the array dynamic here (the ALLOCATABLE attribute). The standard solution is to declare a private array of pointers. Fixed size memory chunks are allocated and assigned to the next free pointer, when the previously allocated memory is full. Since pointer arrays are not supported in Fortran 90, the escape is to define a new data type containing a pointer as a member and to declare an array of that type.

Note, that the data type particle_record and the function to fill in the particle array realize a simple form of data abstraction.

3.2 Data Abstraction

Data abstraction is the way to bundle together (to encapsulate) data and meaningful operations on that data. Data abstraction is a very powerful concept, since is seems to represent more closely a mathematical representation of algorithmic problems. An example of data abstraction for an integer SET data type is shown in ref.[4]. Here we construct an example of a Lorenz vector space.

The intrinsic Fortran 90 operations on vectors and matrices use the euclidian metric. The lorenz metric is defined by the inner product $s = (z^0)^2 - (z^i)^2$, where $z^i, i = 0,\ldots,3$ are the 4-vector components. The operations on 4-vectors should follow the Lorenz metric:

USE lorenz
  TYPE (vector_4) :: a, b, c, p, q, r
  REAL c, s, mass

  p = q ! assignment component by component
  a = c * p ! augmentation of 4-vector by real value
  b = p + r ! addition component by component
  s = p * r ! F77: $s = \sqrt{(p(0)\cdot r(0) - p(1)\cdot r(1) - p(2)\cdot r(2) - p(3)\cdot r(3))$}
  mass = p * p ! i.e. p**2
\[ c = a \cdot \text{boost} \cdot b \] transform \( a \) into the Lorenz frame of \( b \) to obtain \( c \)

The calculus in this example follows closely what is usually written on the paper. The details of the operations are hidden in the module, which is used at the beginning of the program. The functions in the module are called each time the compiler detects the operations on the derived data type which match the argument types in the functions.

Note the special array constructs in this example. Since the vector operations are explicit, compilers will be able to take advantage of the vector machines and the long-instruction-word machines. There are more array constructions available [1] than shown here.

\begin{verbatim}
MODULE lorenz
  TYPE vector_4
    REAL x0, x(3)
  END TYPE vector_4
  INTERFACE OPERATOR(*)
    ! overload the * operator: type of the arguments
    ! determine uniquely the function to be called
  MODULE PROCEDURE vtimesv ! dot product between two lorenz vectors
  MODULE PROCEDURE ctimesv ! augment lorenz vector by a scalar
  MODULE PROCEDURE vtimesc ! idem, other sequence of the arguments
  END INTERFACE OPERATOR(*)
  ! all argument types have been covered
  INTERFACE OPERATOR (+)
    ! add two lorenz vectors
  MODULE PROCEDURE vplusv ! add two lorenz vectors
  END INTERFACE
  INTERFACE OPERATOR (.boost.)
    ! a new operator:
    MODULE PROCEDURE boostme ! lorenz transformation
  END INTERFACE OPERATOR (.boost.)
  CONTAINS
    FUNCTION vtimesv(a, b) ! dot product between two lorenz vectors
    REAL vtimesv
      TYPE(vector_4) :: a, b
      vtimesv = a%\times 0 * b%\times 0 - a%\times * b%\times ! array operation
    END FUNCTION
    FUNCTION ctimesv(c, a) ! augment lorenz vector by a scalar
    REAL c
      ctimesv\times 0 = c * a%\times 0
      ctimesv\times = c * a%\times ! array operation
    END FUNCTION
    FUNCTION vtimesc(a, c) ! operators on derived types are not abelian
    REAL c
      vtimesc = ctimesv(c, a)
    END FUNCTION
    FUNCTION vplusv(a,b) ! add two lorenz vectors
    REAL vplusv
      vplusv\times 0 = a%\times 0 + b%\times 0
      vplusv\times = a%\times + b%\times ! array operation
    END FUNCTION
    FUNCTION boostme(a,b)
\end{verbatim}
TYPE(vector_4) :: boostme, a, b
REAL u(4), v(4), x(4) ! temporary to hold the 4-vectors
u(1) = a%\times 0
u(2:4) = a%\times ! array operation
v(1) = b%\times 0
v(2:4) = b%\times
CALL lorenz4(v,u,x) ! call cernlib routine U101 for simplicity
boostme%\times 0 = x(1) ! but could be trivially rewritten
boostme%\times = x(2:4) ! array operation
END FUNCTION
END MODULE lorenz

This type of programming has also some disadvantages. It is difficult to move between different programming languages, for instance the program above cannot be reused outside of the Fortran 90 environment. Another consideration concerns specifically Fortran 90. Operations have to overload for each combination of argument types, even those that have a trivial conversion (e.g. integer to float; a = c * p in the example above would not work if c is an integer). It is trivial to cover basic types in the operator interfaces. However, a user can define unlimited numbers of derived types that might logically still be handled by the operations already declared, but would not be accepted by the compiler, because they are not mentioned in the interface. For example, consider creating a new type called named_vector_4 with the same properties as vector_4, but having a string in addition. Logically, this addition would not change the inner product, but all the operations have to be redefined. In Fortran 90 this problem can not be solved in general and caution must be applied in creating new data types with operators.

In the object oriented languages this problem is solved by the inheritance mechanism, discussed in chapter 5.

3.3 Library interfaces

To illustrate a possible way to use libraries in F90 programs we take as an example one of the CERN programming library entries. When a library is defined in Fortran 90, its calling sequences are made explicit with an interface definition and the correct usage is enforced by the compiler:

MODULE kernlib
INTERFACE DS09 ! find the minimum of a function of one variable
SUBROUTINE minval(x, y, r, eps, step, maxf, a, b, f)
REAL, INTENT(INOUT) :: x ! estimate of the abscissa of a minimum of f
REAL, INTENT(OUT) :: y, r
REAL, OPTIONAL, INTENT(IN) :: eps, step, a, b
INTEGER, OPTIONAL, INTENT(IN) :: maxf
INTEGER, EXTERNAL :: f ! user function to be minimized
END SUBROUTINE minval
END INTERFACE
END MODULE

This routine can be used in a variety of ways:
call minval(x, y, r, myfunc)
call minval(x, y, r, eps=1.e-6, myfunc)

etc.

There is a possibility to create optional arguments (attribute OPTIONAL). In the body of the subroutine the presence of the optional arguments (e.g. eps) is detected with the statement IF(PRESENT(eps)) ...

There is no way to enforce the correct calling sequence of the user function f at compile time. This is due to the fact that Fortran 90 does not have the notion of a pointer to a function. The compiler cannot check if myfunc has in fact the correct type and number of arguments in the user's program.

However, the calling semantics of the user function f can be checked at run time with the following piece of code in the body of minval:

USE kernlib ! include the interface for compilation of the minval code
SUBROUTINE minval(x, y, r, eps, step, marf, a, b, f) ! subroutine body
... (entity declarations as in the interface, including the external function)
INTERFACE f
   INTEGER FUNCTION f(ifu, xfu)
   INTEGER, OPTIONAL, INTENT(IN) :: ifu
   REAL, OPTIONAL, INTENT(IN) :: xfu
END INTERFACE f
C executable code follows
IF(.NOT.PRESENT(ifu)) CALL error
IF(.NOT.PRESENT(xfu)) CALL error
... (etc., doing the actual function minimalization)

Note that since the interface to a library entry is defined explicitly, the actual call can be replaced by a stub call (e.g. with a preprocessor), making the processing on a remote machine transparent for the user.

3.4 Preliminary conclusions on Fortran 90

The Fortran 90 constructs allow us to write code in a modern programming style, making the programs more powerful and more reliable.

In addition, FORTRAN 77 programs will compile under Fortran 90. There are some restrictions on mixing the constructs of the two languages in the same program. However, there will be F77/F90 translators to make the transition to the next standard easier.

Nevertheless, Fortran 90 is not a universal programming language:

a) Fortran 90 pointers are not usable for low level and on-line programming. There are no pointers to functions – a prerequisite for error recovery and asynchronous processing.

b) Except for some (optional) status variables there is no error handling.

c) There is no asynchronous processing.

d) There is no inheritance (see next chapter).

e) Fortran 90 is too late for the interface builders based on X. The Fortran 90 compilers from major vendors are expected in 1992/93 (IBM, DEC, HP); Cray promises for 1991, but in general the compilers will arrive in 1 to 3 years from now. The development of user interfaces based on the X-Window protocol definition has already started (using C/C++).
f) Fortran (90 and 77), by itself, is intrinsically incompatible with UNIX OS.

As pointed out in chapter 2, a special binding standard is required to be able to access the OS functions from Fortran programs. It is expected that the same working group that has standardized the FORTRAN 77 UNIX interface will do the Fortran 90 UNIX bindings in 1991/92. This will dramatically improve the Fortran/UNIX situation, but the points a) to e) still hold. The main conclusion here is that if Fortran 90 is adopted by the HEP community, several languages (at least two) have to coexist (i.e. for the off-line and the on-line programming; for user interfaces). Therefore, the following point is important in this discussion:

\[ g) \text{ The Fortran 90 standard does not define interfaces to other languages.} \]

4 C++: an (object oriented) extension to C

The definition of the C++ programming language comes from Bjarne Stroustrup [5]. It is (almost) compatible with the ANSI C programming language definition in the sense that an ANSI C program can be compiled by a C++ compiler. In fact, many C++ compilers are just preprocessors to a C compiler. C++ has the advantage of much more rigorous code validation by the compiler, as compared to plain C. The C programming language is popular in the on-line programming, since it allows pointer manipulation and asynchronous processing. The additional features of C++ make it a serious candidate also for off-line programming.

The following strong points of Fortran 90 apply also to the C++ language:

- derived data types
- operator overloading
- internal procedures
- module (object) interfaces

The examples of the Fortran 90 code can be trivially rewritten in C++ semantics. There is a difference in operator overloading: Fortran 90 allows the creation of new operators .op.; C++ can overload only existing operators. On the other hand C++ overloads array references and pointer dereferences (the so called "smart pointers"). Also the notion of a pointer to function is very powerful in C++.

The ANSI C and C++ have the concept of the function call template, therefore floating operations do not have to be converted to double, as it is done in plain C. This makes the floating point arithmetic more efficient. C++ does not have a powerful set of mathematical functions defined in the language, these have to be supplied from outside, but can be made indistinguishable from the intrinsics. A programmer's effort must go into optimizing the library, since the vector constructs are not explicit for the compiler.

The main difference between C++ and Fortran 90 is the presence of the inheriting class concept in the former.

4.1 The standardization of C++

A new committee has been formed under ANSI X3 to standardize the C++ programming language. The first meeting of this committee will take place in spring 1991.

Most features of C++ version 2.0 are already quite stable and will presumably be fixed as they stand today. There might be exception handling and parametrized types in addition.
5 Application of Inheritance in HEP

Data abstraction is a very powerful concept. It can be limited, however, by the necessity to write out all possible usage of the operators explicitly and by the proliferation of the data types. Thus, the next step is to define an abstraction of the data abstraction. This step identifies the collection (classes) of objects, grouped by their operations (behaviour). The proliferation of operations is called inheritance. From abstract classes concrete classes are built in class hierarchies. Workable systems have been already demonstrated in the HEP environment, but they are still not large enough to convince everyone of the validity of this approach.

Since the classes are grouped by their common behaviour, the behavioural analysis is instrumental in the analysis of the system[6].

The analysis of the off-line software was concentrating so far on data modeling. It seems, that the data modeling (e.g. Entity-Relationship analysis) is only a part of the story. It is equally important to realize how the data is going to be used. This approach naturally leads to the data abstraction. It is possible to group data by their behaviour, therefore it leads also to the abstraction of the data abstraction.

At the conceptual level, there seems to be no escape from this approach. If so, the programming language of chose should allow for the inheritance to be able to take advantage of abstract classes.

More practical research is needed to validate this approach.

6 Conclusions

The computing environment of the next generation HEP experiments will consist of network of workstations running UNIX. The Fortran 90 environment is adequate to program the off-line analysis systems. Moreover, the layer of libraries necessary in the present systems programmed in FORTRAN 77 will be reduced and otherwise can be made looking like a native Fortran 90 system with the data abstraction mechanism. However, if the abstraction of the data abstraction is found to be practical, next step to an object oriented language like C++ is necessary.

References


