Accredited Standards Committee
X3, INFORMATION PROCESSING SYSTEMS*

DOC NO: X3J16/90-0095
DATE: November 9, 1990
PROJECT: Programming Language C++
REF DOC:
REPLY TO: Jonathan E. Shopiro

What the Standard Standardizes

Abstract

A programming language Standard must cover a variety of topics, including the language itself, processors, programs, libraries, etc. This paper considers what their status should be in the Standard, and offers guiding principles for the writing of the Standard.

The purpose of this note is to clarify the point of view of the C++ Standard. A problem with other programming language standards is that it isn't always clear what is being standardized, what is a requirement, and to what the requirements apply. Making these things clear will make the C++ Standard more precise and easier to use.

A programming language Standard, or any work covering all aspects of a programming language, must be concerned with many things: the language itself, programs written in the language, implementations of the language, libraries, and environments, to name a few. A Standard, however, has a special function. It is a testing or measuring device. It tests samples of some kind of product and determines (only) if they do or do not *conform*. The Standard contains a set of requirements and samples conform to the Standard if and only if they satisfy the requirements. A Standard is good if it is unambiguous and easy to use and if it establishes the right criteria (the right samples are selected or rejected).

Thus, the first thing to know about a Standard is what kinds of things it tests. In the case of the C++ Standard, the three obvious choices are the C++ language, C++ processors (processor is the ISO terminology for the totality of an implementation: syntax checker, compiler, interpreter, libraries, run-time system, hardware, everything), and C++ programs. It is certainly possible to have a Standard that tests more than one kind of thing (e.g., bolts and nuts), but a Standard is likely to be simpler in structure if it only tests one kind of thing.

It is clear that the Standard must establish requirements for C++ processors. There are requirements that only apply to processors. For example, a processor must say what the minimum and maximum values of an int are. In general, "implementation defined," as defined by the C Standard, means that the processor is required to say what it does. Obviously, the most important requirement on C++ processors is that they implement some version of, or approximation to, the C++ language. It is the job of the Standard to say what deviations from the language

* *Operating under the procedures of The American National Standards Institute (ANSI)*
Standards Secretariat: CBEMA, 311 First St. NW, Suite 500, Washington, DC 20001

are allowed. For example, implementation limits such as the minimum depth of parenthesis nesting that must be supported are most naturally thought of as a requirement on processors, or as a specific relaxation of the global requirement that they implement the C++ language.

It is probably most reasonable to think of the C++ language itself as a mathematical object which executes on an abstract machine (or actually a range of mathematical objects; see note below) and not something to be tested by the Standard. Obviously the Standard needs to define C++, and that will be the bulk of the Standard and of our efforts. *It is not necessary, however, to write the C++ Standard as a standard for languages that tests whether or not a particular language is C++.*

This will have a significant impact on the Standard, because the ISO recognizes that the user of a Standard needs to be able to identify the requirements that a conforming sample must fulfill and therefore it mandates special, rather stilted language for requirements in Standards. We can structure the C++ Standard so that it defines the C++ language but only *specifies* the way and the extent to which a C++ processor shall implement the C++ language. From the requirements point of view, the definition of C++ is an auxiliary, but still essential, part of the Standard. Then the language definition can be done in ordinary (but precise!) mathematical and computer science language. The stilted normative language need only be used in those places where a processor is allowed to depart from the language or there are extra-language requirements on the processor. Otherwise we will figuratively have to fill a pepper shaker with *shall's* and apply it liberally to the document.

If the C++ Standard defines the C++ language and specifies C++ processors, it need not say anything more about C++ programs. That is, any question that one might reasonably ask about C++ programs can already be answered. Will a certain program be accepted by all conforming processors? Will it be rejected by all conforming processors? Will a certain program execute the same way on all conforming processors? These and all similar questions can be answered by such a Standard, because the Standard says what processors are required to do and what they are allowed to do. This avoids the necessity of trying to distinguish between conforming and strictly conforming programs as in the C Standard.

In practice, a Standard structured in this way will be much like the draft that we have now, because the draft mostly describes the C++ language. Wherever the Standard must specify a requirement on C++ processors, beyond the global "A C++ processor shall implement the C++ language," it would probably be done in a special section, perhaps entitled "Processor Requirements." Issues such as data representations and structure layout would be handled in those sections.

Structuring the Standard as described above will have perhaps the least effect on users — both language implementors and programmers — except that the Standard will be clearer and more precise than a Standard that doesn't have a consistent point of view.

The keyword volatile is an interesting test case for the idea of defining the language and specifying processors. It seems to me that volatile's place in the language is strictly syntactic (following the ISO terminology, which lumps what we sometimes call static semantics with syntax) but that it has an important meaning to processors, which are required to make operations on volatile objects apparent.

Casts are another interesting case. Currently we have frequent controversies over exactly what should happen when one kind of object is cast to another (other than user-defined casts). I am not sure, but I suspect these controversies will be easier to resolve if we can decide on what the *language* should do separately from whether processors should be required to fully and exactly *implement* that aspect of the language.

Libraries in the Standard can most easily be defined as mathematical objects, it seems to me, with issues such as mathematical accuracy specified as requirements on processors.

Environmental issues, on the other hand, are mostly outside of the language and will have to be dealt with in the Processor Requirements sections.

Note: What Kind of a Mathematical Object is the C++ Language?

There is a lot of slack in the definition of C++ in the ARM, and we want to leave a lot of slack in the Standard. There are two ways to represent this concept in a mathematical definition. One is to say that C++ is a collection of languages (for example one language has 32-bit ints and another has 16-bit ints, or one language evaluates function arguments left to right and another evaluates them right to left). Then a processor would be free to implement any instance of C++. It would have to say (to the extent specified by the Standard) which instance of the language it implements.

The other way to accommodate slack is to describe C++ as a non-deterministic language. Then at certain points in the execution there are choices to make and a processor is allowed to make them arbitrarily, or is required to say how it makes them, as specified by the Standard. The Standard may even allow the processor to do something strange in those cases where the decision would have an impact on the state of the program.

I think the best approach is to use both techniques, that is, define C++ as a collection of non-deterministic languages. Generally the processor would be required to say which instance of the language it implements but would not be required to say how it makes choices, so those items which are considered implementation-defined would engender different versions of the language, and the unspecified items would be represented as non-deterministic choices.