FOREWORD: Serving the C++ Community

The ISO standard is the culmination of the early years of C++ and the basis for its further evolution. This foreword examines the role of the standard committee in the life of C++ and ponders the future of Standard C++. C++ is a general-purpose programming language with a bias toward systems programming. By design and necessity it is a language that can — and often does — provide run-time performance unmatched by anything but carefully handcrafted assembly code. Similarly, by design and necessity it is a language that is available on a wide variety of platforms from some of the tiniest processors to some of the most ambitious supercomputers. In addition, C++'s abstraction mechanisms — such as classes and templates — effectively serve users for whom high performance is merely a fringe benefit. The result has been an extraordinary diversity of application areas and user communities.

This widespread and diverse use implies centrifugal forces that could tear the language apart into competing dialects, thus destroying the portability and shared community that are among the most valuable benefits of a common general-purpose language. This danger is made more acute because C++ lacks conventional centralizing forces such as a corporation with legal ownership (and a deep war chest for marketing and development), a dominant implementation, or a centralized users' organization.

The C++ community's response to this challenge was to establish the ISO C++ standards committee (and its national counterparts) and to support its work for over a decade. The honor of taking the initiative goes to people from DEC, HP, and IBM, who — led by Larry Rosler and Dmitry Lenkov — in late 1988 contacted me about starting ANSI standardization of C++. It took them a couple of hours to convince me, but once convinced I started to prepare. The ARM (''The Annotated C++ Reference Manual'') was the first and most visible result. The ARM, or rather the reference manual in ''The C++ Programming Language (2nd edition),'' became the base document for the ANSI standardization of C++. AT&T deserves special praise for contributing this reference manual to the standards effort. A less enlightened company might have tried to retain C++ as a proprietary language.

The first and most significant effect of forming the ANSI C++ group was as obvious as it was unexpected to people unfamiliar with standardization. Suddenly, the community had an open forum where people from different parts of the community could meet to discuss all topics of importance to the language and its users. Commercial allies could talk without falling foul of cartel rules, commercial competitors could settle technical issues without getting into trouble with corporate rules, and there now was a place and time for people to meet face to face. In the committee, I finally met people I had known for years via email.

The C++ standardization always involved people from many countries. Even the ARM was reviewed by more than 100 people from half a dozen countries, and the first organizational meeting of what was to become the ANSI C++ committee (in Washington D.C. in December of 1989) had participants from several countries. In 1991, this international aspect of C++ was formalized by making the standardization an ISO process.

In retrospect, the committee's role as an open forum and as a focus of the C++ community was at least as important as the final standard. The C++ standards committee is the center of the C++ community. Had the members not left their commercial rivalries at the committee door and focused on technical issues, the C++ standards effort would have failed. I am told that this consistently civil, civic, and often friendly behavior is uncommon in a standards committee. One of the things that have made the C++ standards process worth while for me personally and professionally is that only in the rarest of cases have members interpreted their obligations narrowly or destructively. I cannot think of a meeting where I didn't learn something worthwhile. The C++ committee has been the source of many long-term friendships.

The committee membership was always large enough and dedicated enough to keep alive communication with the community at large. The image of a handful of experts meeting in some ''isolated, small, dark, smoke-filled room'' is about as far from the reality of the C++ committee as you can get. I once

From The Wiley publication of the C++ Standard(ISO/IEC 14882:2002). ISBN 0 470 84674-7.

counted 102 people in the meeting room. There are rarely less than 50 people attending a meeting and something like 200 people participate between meetings using email and websites. I have seen people providing real-time reports to newsgroups from the meeting room. In addition, several nations have their own groups and meetings. From this extended group of members, information flows to and from the rest of the C++ community through email, newsgroups, presentations, papers, and books. This open, two-way, communication has been essential for the development of C++.

The result of the committee's work so far and the basis of all future work on C++ is the current standard. Code written against this standard will work for years and even decades to come. The virtues of the standard document are precision and detail. For most readers, its problems are that it is hard to read, contains no rationale, and gives only few hints at implementation techniques. The standard is not a tutorial, not even a tutorial for experts. There are textbooks covering that need, and people who try to learn C++ from the standard usually have a terrible time figuring out the language features and — like anyone trying to learn C++ from a manual — fail to appreciate the programming techniques those features exist to support. The closest thing we have to a rationale for C++ is my ''The Design and Evolution of C++'' which discusses the motivation behind most of the major design decisions.

A standard is just a heap of paper (or a bag of bits) until someone takes it seriously enough to provide implementations that approximates it as closely as can be done given real-world constraints. The C++ community is now fortunate enough to have several high-quality implementations that closely approximate the standard and at least one that passes all conformance tests with flying colors. These implementations and their numerous ports cover more platforms than I could possibly know of. The community owes thanks to the compiler and standard library writers — many of whom are members of the committee or represented there by colleagues. Standard C++ is not an easy language to implement, yet for a given platform there are often a choice of good compilers and standard library implementations. One implication of this wide availability is a high degree of practical portability (if that's your aim) across an astounding range of systems.

The committee was never just backwards looking. Naturally, much of its work is concerned with the detailed, tedious, and necessary work of clearly stating existing rules and resolving compatibility problems. However, a living language cannot be static and unchanging, so C++ has evolved to meet challenges. For example, in my 1990 C++ reference manual, exceptions and templates were labeled ''experimental,'' yet they were obviously essential parts of what C++ was supposed to be. Today, through the work of the committee, the efforts of implementers, and the creative ideas of users, templates and exceptions are the key to some of the most powerful and innovative uses of C++.

Library development has always been one of the most vital driving forces of C++. One of my earliest principles for C++ was to prefer facilities that ease or enable library building over facilities that merely solve specific problems. This idea was sorely tested by the lack of a good standard container library. C++ was expressive enough to define a good string class, a good complex number library, and good specific vector and list classes, but by 1992, nobody had come up with a sufficiently general, flexible, and efficient container framework. Most ideas were derivatives of the original Simula containers-of-objects design. However, such containers cannot hold built-in types or classes not specifically derived from the container element base class (usually called ''Object'' or ''Link''), require ugly and inefficient casting when retrieving objects from a container, leads to overuse of heap storage, and lack any effective notion of container interchangeability; see the discussion in chapter 16 of ''The C++ Programming Language (3rd edition).'' Clearly, templates held the key to a better solution, but until Alex Stepanov came forward with his STL, nobody had constructed a sufficiently general, elegant, and efficient solution to the standard container problem.

The STL evolved into the containers, iterators, and algorithms part of the C++ standard library. Through that, it became the inspiration for a generation of C++ programmers and the starting point for modern generic and generative programming. This had repercussions beyond the C++ community. The success of templates and STL-style containers have been one of the major reasons for the inclusion of simple generics into Java and C# and for the work on comprehensive support for generic programming in dialects of functional programming languages such as Haskell and ML. In this, the impact of C++'s generic programming facilities and techniques resemble the impact that C++'s object-oriented programming facilities and techniques had on the programming community as a whole a decade before when C++'s mainstream acceptance helped inspire CORBA, COM, Java, and even object-oriented facilities for Fortran and Cobol.

So where does the C++ standards process take us from here? The current standard describing both

From The Wiley publication of the C++ Standard(ISO/IEC 14882:2002). ISBN 0 470 84674-7.

language and library is a better approximation of my original aims for C++ than any previous version. However, we can obviously do better still and the standards committee intends to live up to that challenge. Unfortunately, there are things that a committee cannot do well. Among the most obvious of those are innovation and design. The current standard contains a few obvious and embarrassing examples of design by committee in both the core language and in the standard library, but I consider those blemishes, potentially correctable, and non-critical. As the work on the next standard begins, the committee must and will try even harder to avoid that trap.

The most successful additions to C++ were developed outside the committee and appeared as well-developed ideas combining features and a philosophy of use. The best example is again the STL. In contrast, less successful developments tend to be minor technical enhancements that grew as the committee tried to compensate for a lack of clear vision. Examples are std::string and some of the details of templates. What the committee seems to do well is to refine facilities given a clearly articulated vision and concrete code examples.

What directions might the committee take? That's for the committee to decide, so I can only guess and I hope that we'll again be surprised by some brilliant new insight or library that can disrupt the regular proceedings and set C++ on a new and more productive track. However, in the absence of such an exciting breakthrough, the predictable directions are Improve the support for generic programming Improve the support for library building and use Make the language more regular, predictable, and teachable Standardize libraries to make C++ a better platform for systems programming

These directions will eventually become reasonably well specified and backed by specific language proposals, library proposals, explanations of programming techniques, and concrete examples. Ideally, the size and number of core language extensions will be low because facilities provided as libraries can be more easily and more thoroughly tested before their inclusion in the standard. Compatibility with the current C++ standard, improvements in the degree of static type safety, and adherence to the zero-overhead principle (''what you don't explicitly ask for, you don't pay for'') are considered very important.

These directions affirm the original guiding principles for C++, distinguish C++ among programming languages, and will serve the diverse C++ community well. Bjarne Stroustrup

Texas A&M University and AT&T Labs
bs@cs.tamu.edu