http://www.devx.com Printed from http://www.devx.com/SpecialReports/Article/38813/1954

The State of the Language: An Interview with Bjarne Stroustrup

C++ founding father assesses the language on the eve of its new standard.

by Danny Kalev

Bjarne Stroustrup, inventor of the C++ programming language, is a computer scientist and the College of Engineering Chair Professor of Computer Science at Texas A&M University. He has always been highly involved in the standardization of C++. Since 2004, Bjarne and the standards committee have been busy hammering out the details of a new standard, temporarily titled C++0x. Bjarne was gracious enough to take time out of his busy schedule to speak with DevX C++ Pro, Danny Kalev, about new C++0x features and the state of the C++ language.

The C++0x standard will be finalized during 2009. Can you outline its major features and its overall importance to the C++ community?

We hope to vote out a draft standard for public review in October 2008 so that we'll be able to hand in a final draft for international vote in 2009. Because of this heavy ISO process, it's touch and go whether C++0x will be C++09, but there is still hope and the major features are now known (barring disasters). We can classify the extensions like this

Concurrency:

- memory model supporting modern machine architectures
- Threading ABI
- atomic types
- mutexes and locks
- thread local storage
- asynchronous message exchange

Libraries:

- regex: regular expressions
- unordered_map, etc. (hash tables)
- smart pointers
- array: fixed-sized array
- improvements to containers based on new C++0x features
- tuples
- date and time (maybe)
- various library components to held library builders

Language:

- rvalue references (move semantics)
- static_assert: static assertions
- variadic templates
- strongly typed enumerations with scoped enumerators
- constexpr: generalized constant expressions
- control of alignment
- delegating and inheriting constructors
- auto: deducing a type from an initializer
- decltype: a way of using the type of an expression in a declaration
- control of defaults

- nullptr: a name for the null pointer
- a range-based for loop
- lambda functions
- raw string literals
- UTF8 literals
- concepts (a type system for template arguments)
- initializer lists and uniform initializations syntax and semantics
- in-class member initializers

Lists are, by themselves, not very interesting, but you can read up on a description of my general philosophy for language evolution and some of the individual decisions in my HOPL-iii paper "Evolving a language in and for the real world: C++ 1991-2006." You can also find more information than you could possibly want on the committee's web site.

Basically, the "concurrency" features will standardize the basic layers needed to do systems programming in a multi-core world. Obviously, facilities for doing that already exist in C++ implementations, but they are not standardized. I'd have liked to see library support for some high-level concurrency models, but the committee didn't have the time or consensus for that.

The library facilities provide a set of new library components and some improvements to the existing ones. I would have liked to see many more, but the committee is a volunteer effort and we just didn't have the resources for a massive extension of what was offered. Fortunately, there are many libraries available "out there," possibly already on your machine. For example, many of the C++0x libraries (e.g., regex and unordered_map) are now shipped by major vendors and boost.org offers many components (for instance, file system and networking) that we'll probably soon see in the standard. There is also much talk of a technical report on the libraries we most wanted but had to postpone.

"I'd have liked to see library support for some high-level concurrency models, but the committee didn't have the time or consensus for that."

The language extensions are a varied lot. Fortunately most are small and fit together with each other and with existing facilities to make a better integrated language. Consider a few examples:

```
// using C++0x features:
vector<string> v = {"Gorm", "Harald", "Sven",  "Harald", "Knud" };
for (auto p = v.begin(); p!=v.end(); ++p) cout << *p <<'\n';
for (auto x : v) cout << x <<'\n';
```

I suspect most is pretty obvious. You can provide an initializer list directly for a vector, you can deduce the type of an iterator from its initializer, and iterate through a sequence without explicitly mentioning the iterator. More examples:

```
enum class Season { winter, spring, summer, fall };
int summer;                 // doesn’t clash with Season::summer
Season s = Season::spring;  // note qualification
summer = s;                 // error: no Season to int conversion
Season += Season::fall;     // error: can’t add Seasons

for_each(v.begin(), v.end(), [](const string& s) { cout << s <<'\n'; });
```

You can have enumeration types (class enums) that behaves more like types than glorified integers and the “lambda” notation ([], etc.) is a simplified notation for defining a function object. Each of these simple examples can without heroic effort be written today. However, in C++98 that code would be twice the size and in every case open opportunities for making errors and/or for introducing overhead.

Note that all "primitive" features are meant to be used in combination and in combination with existing features to solve problems. For example, there is no "magic" for initializing vectors in particular or even any new "magic" for constructors. Instead, the initialization of vector was achieved simply by a rule that state that an initializer_list<T> can be initialized by a list of any numbers of Ts {t1, t2, t3} for any type T. Given that rule, we simply give vector<T> a constructor that accepts an initializer_list<T>. Incidentally, this mechanism can also be used to eliminate about 80 percent of the uses of the type-unsafe stdargs macros.

Q Concepts are probably the most important addition to C++ in years. Can you please explain which problems in contemporary C++ concepts will solve?

A I suspect that what is most important depends on what you consider important. For many, concepts will be invisible. They will simply be "the magic" behind a radical improvement of the error messages you get when you make a mistake using something that happens to be a template. Basically, you'll get the kind of error messages that you are used to from "ordinary functions" and at the time you get them for mistakes with ordinary functions. What's so great about that? I can imagine a time a few years into the future where people won't be able to think of a good answer to that question. This will be the ultimate proof of the success of concepts.

"Concepts" is a type system for types, combinations of types, and combinations of types and integers. Basically, it allows the programmer to state in code what we currently state in documentation and comments. For example, the standard algorithm takes a pair of forward iterators and a value; given that, it assigns the value to each element of the sequence defined by the iterators. That algorithm's requirements on its argument types can be expressed like this:

```
template<Forward_iterator For, class V>
       requires Assignable<For::value_type,V>
void fill(For first, For last, const V& v);   //  just a declaration, not definition
```

We can try using fill():

```
int i = 0;
int j = 9;
fill(i, j, 99);       // error: int is not a Forward_iterator

int* p= &v[0];
int* q = &v[9];
fill(p, q, 99);       // ok: int* is a Forward_iterator
```

The checking of use happens immediately at the call site and uses only the declaration. This is a major practical advantage over the current state of affairs for templates where the checking happens late—during template instantiation time.

Concepts also allow us to check the definition of a template in isolations from its arguments:

```
template<Forward_iterator For, class V>
       requires Assignable<For::value_type,V>
void fill(For first, For last, const V& v)
{
       while (first!=last) {
              *first = v;
              first=first+1; // error: + not defined for Forward_iterator
                             // (instead: use ++first)
       }
}
```

Again, the error is immediately caught at the point in the definition where it happens.

This checking is the most obvious benefit of concepts, but they also give greatly improved specification of algorithms and class templates and makes overloading straightforward, eliminating the complicated scaffolding of traits and helper functions that people have to rely on today.

Most importantly, concepts help us think about our algorithms and their requirements on their (template) arguments. This leads to more reliable and more general code. Often, such thinking also leads to improved performance as we are forced to think about what the algorithm really needs and what is extraneous.

Q Let's talk about rvalue references. Will they also be "mostly invisible" as concepts, allowing library writers to implement perfect forwarding and move semantics or will they affect the average C++ programmer's code more visibly? For instance, will C++ have a fifth canonical member function called a move constructor?

A Yes, rvalue references are very much a "technical extension" meant to be invisible to 99.9 percent of programmers. Their main effect will be some speedup in the implementation of common containers (e.g. vector) and algorithms.

The basic idea behind rvalue references is that we often copy a value when all we wanted was to move it; that is, after a=b, we have two copies of the value of b, but often the very next thing we do is to destroy b. With rvalue references, we can write a function move() so that we can write a=move(b) to express that idea. If b is something large (e.g. a string or a matrix), the performance advantage can be significant (i.e. a factors rather than a small percent). Such move operations will be sprinkled around in the standard library implementation so that you gain the performance advantage even if you have never heard of rvalue references.

For example, some standard library classes (e.g. string and vector) will have "move constructor":

```
T::T(T&&);
```

which can basically be read as "T doesn't waste time making copies where it can move."

Q In your recent SD West lecture, you predicted that "concepts are going to be sexy for awhile, then overused and then people will get sick of it. When things calm down, we'll find out when and where they're actually useful." Where do you draw the line between valid, welcome usage of concepts as opposed to "overuse"? In other words, what shouldn't we do with concepts?

A I'm pretty sure that was a comment about generic programming in general, rather than about concepts in particular. I'm hesitant to draw a sharp line. Doing so would imply that I thought I knew what would be right (and not) for essentially all people in essentially all application areas. I don't, and drawing such lines is also a sign of paternalistic language design. You can "draw lines" for individuals and organizations (that's what coding standards are for), but not for a set of general language features for a general-purpose language. The point about "general" is that programmers will discover techniques beyond the use cases that the language designers thought of. When a new and powerful feature becomes available, it gets overused as adventurous programmers try to push it to the limit. In the early days of C++, we got inline virtual overloaded functions in multiply inherited classes. When templates were new, we got excessively clever template meta-programming. This will happen again with the C++0x feature set. I just hope that this time more people can tell the difference between an experiment and something ready to put into production code.

"Actually, many of the C++0x features, including concepts, are there to ensure that less cleverness is needed to express interesting things and to make what is expressed better checked."

Actually, many of the C++0x features, including concepts, are there to ensure that less cleverness is needed to express interesting things and to make what is expressed better checked. Naturally, that will free up time and energy for even more extreme/interesting experiments.

What shouldn't we do with concepts? I don't know, but first, I'd use them to precisely and exhaustively express the requirements of generic algorithms on their (template) arguments. That should keep me busy for a while. For starters, there is all of the Standard Library (being done by the committee and in particular by Doug Gregor and friends) and all of classical mathematics. Once that is done, we should have the experience to do a good job at less regular or more complicated uses of templates. However, please remember that the aims are simplicity, regularity, and performance. You don't prove that you are clever by producing the most complicated code.

Q Is your prediction regarding concepts based on what happened with meta-programming? What is your stance regarding meta-programming in general? Does this paradigm really offer something indispensable or is it sheer "cuteness" that has been overstretched?

A Partly, but more directly my guess is based on what happened with class hierarchies and overloading. Every powerful new language feature or programming technique goes through a period of overuse. Later, when understanding is better diffused in the community, the more reasonable, effective, and maintainable uses become standard practice and the "interesting" uses get relegated to experimental uses. I think that there are uses of template programming that go beyond both "classical generic programming" and "cute." I think a first attempt of those would be generative programming where the template meta-programming techniques are used to generate pretty straightforward code, possibly generic code.

Q Would it be correct to say that concepts make type traits less needed (or even redundant) in C++0x? Or do type traits still offer capabilities that concepts don't have?

A Concepts can and should eliminate most uses of traits, helper functions for dispatch, and SFINAE overloading.

Basically, the relationship between concepts and traits is that concepts make overloading simple and straightforward without scaffolding (including traits). For example, assuming a suitable set of concepts we can write:

```
template<RandomAccessIterator I>
requires Comparable<I::value_type>
void sort(I b, I e);

template<Container C> void sort(C& c) { sort(c.begin(),c.end()); }

template<Container C, Callable Cmp>
       requires Callable<Cmp,C::value_type>
void sort(C& c, Cmp cmp) { sort(c.begin(),c.end(), cmp); }
```

Picking the right sort() is trivial:

```
vector<string> v;
// …
sort(v);
sort(v.begin(),v.end());
sort(v,Not_case_sensitive());
```

There are uses of traits that do not relate to overloading, such as character_traits, which will remain after concepts become universal. However, most of the prominent uses of traits will be eliminated.

Q Have you had a chance to program in any of the newer programming languages (Java, C#, Ruby, and so on)? Do you find in them anything that impresses you or worth commending in terms of novelty, engineering merits, or simplicity?

A I have tried a lot of languages, including those you mention. I'd prefer not to do language comparisons. Such comparisons are rarely fair and even less frequently seen to be fair. However, "simple" is not the first word that springs to mind. I note the size of the run-time support involved. I predicted the growth of Java complexity and don't condemn it—I consider complexity an inevitable consequence of serving a large community. The design aims of those languages are not those of C++ and vice versa.

Q I feel that there is pressure to add to C++ features such as finally, garbage collection, and dynamic array bounds checking as an attempt to appease users of other languages. In reality, these features don't fit well into the design aims of C++: finally can be replaced by RAII and local classes anyway, GC and destructors are mutually exclusive, and runtime bounds checking violates the pay-as-you-go and trust-the-programmer principles. Does C++ truly need a GC? And more generally, where do you draw the line between borrowed features that are indispensable and those that are not?

A There is always pressure to add features. Many people think that I and the committee are just bloody-minded and/or stupid not to immediately add their favorite feature—typically a feature they have tried or just heard of in some other language. Often, those same people complain that the committee is adding too many features and that we should remove some of those old and ugly "legacy features." Making changes to a widely used language is not easy. There are distinct limits to how many changes we can add with a reasonable hope that they will be widely useful, rarely harmful, or confusing, and not breaking existing code. People really don't like their existing code to be broken and making significant extensions 100 percent compatible and properly integrated with all existing and new features can be quite difficult.

I don't know how to draw a sharp line between worthwhile and frivolous extensions, but I do know that no new feature is really "indispensible." I try to evaluate each new suggested feature on its merits and in the context of existing language features, library facilities, known problems, and other proposed features. Since the number of new features we can accept is limited, I try to maximize utility, minimize damage, and minimize implementation cost. Each new language feature is an intricate puzzle and the more fundamental a new feature is, the more parts of the existing language and existing usage are affected and must be taken into account. For example, the new strongly typed enums were relatively easy to design and implement because they are an isolated feature, but conversely they are also unlikely to have a dramatic impact on the way people design their code and view C++. On the other hand, the new facilities for initialization are likely to impact every user and be highly visible in code. On the other hand, their definition touches many points of current usage and definition and is therefore at least an order of magnitude harder to design/define than the enumerations.

finally clauses for try blocks are a minor issue and—as you mention—redundant in that we can use RAII. It can even be seriously argued that providing finally would lead to uglier and more buggy code as programmers used to it in Java (or whatever) found it easier to avoid learning RAII. finally is not on the list of C++0x features.

The garbage collection issue is not simple. I even think that your question oversimplifies the issues—we need to find a way to combine GC and destructors. We will not see GC in C++0x, but we will see further work on a design for optional and programmer-controlled GC. For C++0x, we will get a definition of what it means for a pointer to be disguised and an ABI for deeming areas of memory "not containing pointers" and/or "cannot be collected." The result of these simple guarantees will be that existing add-on collectors will be more portable and more effective.

First, let's clarify the ideals: We want simple and comprehensive resource management. That is, we want every resource acquired to be released ("no leaks"). To get that, we need a programming model that is simple to use. A complicated model will lead to errors (leaks) when people misuse it or give up on it in favor of (often even more error prone) "home brew" solutions. "Comprehensive" is essential because memory isn't the only resource we need to worry about; we need to handle locks, sockets, file handles, thread handles, etc.

We have RAII (for scoped resource use) plus "smart pointers" (for resources that don't have their lifetimes determined by a simple scope) as a comprehensive solution. From this perspective, the smart pointer types in C++0x completes the RAII technique supported in C++98. Unfortunately, this set of techniques works only where people use it systematically and correctly.

For example:

```
void f(int i, char* p)
{
vector<X> v(i);
string s(p);
// …
}
```

Here the storage for elements of v and x are handled automatically by the destructors of vector and string. If X has a non-memory resource (for instance, a lock) vector's destructor will release it. This style of use is simple, widely understood, and efficient.

The reason that GC is attractive to me is that there are projects where I think that "RAII plus smart pointers" are unlikely to be systematically and correctly used. Examples are projects where exceptions cannot be used, projects with a lot of exception-unsafe "legacy" parts, projects with components developed in a number of places with different programming philosophies and programmer skills, and projects with long-established resource management strategies that don't fit the "RAII plus smart pointers" model. Typically, such projects are valuable, long-lived, expensive to rewrite, and they leak. Add-on garbage collectors have been successfully used to deal with such leaks for over a decade. In some cases, the collector is used simply until the leaks can be plugged; in other cases, they are used because someone gave up plugging all the leaks. This use of GC is sometimes called "litter collection" as opposed to uses where programs leak for the convenience of programmers. My guess is that even with the best education based on RAII, we will have programs that need litter collection "forever"—new ones will be written as fast as old ones are made safe.

"I consider it obvious that C++ GC will have to be optional (under some form of programmer control) because some programs do not need GC (they don't leak), some programs cannot afford GC delays (not all collectors offer real-time guarantees), and some programs cannot afford to include a significant run-time support system."

Note that my aim is not to use GC to hide the problems and complexities of resource management, but to use GC as yet another tool for dealing with resource problems. This is quite different from the view of GC as a panacea. The current C++ techniques for resource management deal more directly with the problem than GC and should be used as the first line of defense against resource management problems. One of the strengths of well-written C++ is exactly that it generates so little garbage. This makes GC in C++ surprisingly (to some) efficient.

I consider it obvious that C++ GC will have to be optional (under some form of programmer control) because some programs do not need GC (they don't leak), some programs cannot afford GC delays (not all collectors offer real-time guarantees), and some programs cannot afford to include a significant run-time support system. One of the two major design issues for C++ GC is how to express this programmer control. The difficulty is to ensure that a program component that relies on the absence of GC (for performance, for instance) is never linked with a component that relies on GC. Remember that there is a widespread use of dynamic linking and plug-ins so in many contexts whole-program analysis is not an option.

As you indicate, the other big issue is how to reconcile GC and destructors. If programmers come to rely on GC to collect their garbage for them, they might "leak" an object for the collector to recycle even though that object had a non-trivial destructor—a destructor that releases a non-memory resource. For example, given GC, I might use a

```
vector<X*> v;
```

Without providing code to delete the pointers when v is destroyed if I "know" that X does not own a non-memory resource. Such code would be brittle because a change to X (for instance, adding a lock member), could make my code leak. In general, this problem seems intractable for real-world scenarios involving maintenance of code (adding non-trivial destructors) and dynamic linking. However, my impression is that a combination of explicit declarations of destructors releasing non-memory resources ("explicit destructors") and heuristics can eliminate a high percentage of real problems. Remember that neither of the two "pure" alternatives (no-GC and all-GC) consistently leads to perfect memory management either (in the hands of real-world programmers for real-world problems), so the absence of a perfect solution should not deter us from providing a good one.

I know it's a bit early to discuss the farther future of C++ before the C++0x Working Paper is even finalized. Still, what are the next stages in the standardization of C++ that will take place once C++0x has been ratified?

This question reminds me of coming from an 18-hour flight from Los Angeles into the bright morning light in Sydney to be met by the question "What do you think of Australia?" That was my first trip down-under so my total experience of Australia outside the baggage claim area was about two minutes. That's significantly longer than my post-C++0x experience.

We are planning some TR (Technical Reports) and there is some talk about a faster revision cycle (about three years after the standard). The topics for such TRs (or early revision) are:

Library components:

- networking (sockets, iostreams over sockets, etc.)
- thread pools
- file system
- lexical_cast
- improved I/O (e.g., faster iostreams, memory-mapped I/O)
- special mathematical functions

Language features:

- modules (including dynamic libraries)
- garbage collection (programmer controlled)

And—as usual—a host of little things. I just hope that if the committee takes that path it will not be overwhelmed (distracted) by "little things." As usual, implementations of the library components already exist for experimentation and some (for instance, the networking library) are already in serious commercial use. In particular, see boost.org.

You've been designing and improving C++, along with other devoted members of the standards committee of course, for nearly 30 years. I assume that the addition of templates to C++ in the late 1980 proved to be the most important and successful feature that C++ ever got since you added classes to C in 1979. Is this assumption correct? Is there any C++ feature you regret?

There are things that I would have liked to do better, but who am I to second guess the younger Bjarne? I am no smarter than him and he had a better understanding of the conditions at the time.

In the case of templates, I knew their strengths and I knew their greatest weakness at the time. The first three pages in D&E is a lament for not solving the template argument checking problem. In 1988, I knew the problem, but I don't think anyone knew a solution that would have been viable in the context of the Draconian requirements of flexibility, generality, and performance that were (and are) the bedrock of template use. The design of concepts involved solving genuine research problems; we have papers in POPL and OOPSLA to prove that!

C compatibility has always been a serious problem. The C syntax is horrendous, the conversion rules chaotic, and arrays decay to pointers at the slightest excuse.

These are fundamental and constant problems. However, I chose C compatibility (though never 100 percent compatibility) as a means of making C++ a practical tool rather than yet another pretty language and so we had to live with it. Even today, the overlap between the C and C++ communities and code bases are so large that a serious break of compatibility would simply lead to the language community fragmenting. We have to proceed in the usual way: provide superior alternatives to ugly and/or dangerous features and hope that people use them.

In this respect, I read a Slashdot interview in which you say that you would have anyway started off with an existing programming language as the basis for a new one, as opposed to starting from scratch. Why not start from scratch, really? And why start with C?

Again, I don't have a time machine. At the time, C looked like a good choice for me even though C was not an obvious choice then. My guess is that most people would have chosen something simpler and cleaner, such as Pascal or Modula, over the flexibility and performance of C.

Why not start from scratch? Well, at the time, I decided that to build a useful tool with the skills and resources available, building on an existing language was essential. I wanted a tool, not a beautiful toy. If I had to do a new language, I would again have to evaluate my aims and my resources. Starting from scratch is hard. We always carry our experience with us and anything sufficiently different from the familiar will cause teaching problems. People invariably confuse familiarity with simplicity. Even Java, which was supposedly designed from scratch according to first principles, chose the ugly and illogical C syntax to appeal to C and C++ programmers—and Java doesn't even have the excuse of compatibility. Should I design another language, I guess I would have to try "from scratch" just to get a new set of problems. In that case, I would be very sensitive to the idea that there are other forms of compatibility than source code compatibility, such as link compatibility and source code transformation.

Is C++ usage really declining, as some biased analysts and journalists have been trying to convince us for years (often not without ulterior motives), or is this complete nonsense?

It's probably not complete nonsense. C++ use appears to be declining in some areas and appears to be on an upswing in other areas. If I had to guess, I'd suspect a net decrease sometime during 2002-2004 and a net increase in 2005-2007, but I doubt anyone really knows. Most of the popular measures basically measures noise and ought to report their findings in decibel rather than "popularity." Many of the major uses are in infrastructure (telecommunications, banking, embedded systems, etc.) where programmers don't go to conferences or describe their code in public. Many of the most interesting and important C++ applications are not noticed, they are not for sale to the public as programming products, and their implementation language is never mentioned. Examples are Google and "800" phone numbers. Had I thought of a "C++ inside" logo in 1985, the programming world might have been different today.

Among the positive signs for C++, I can mention an increase in my email and the number of speaking invitations. At the SDWest conference in March, the C++ track was by far the largest, even without counting the "Super tutorial" that I gave with Herb Sutter; more than 300 people attended that one—up by a factor of two compared to two years ago. These are, of course, personal and subjective measures, but they fit with other information that I get from innumerable sources in the industry (worldwide).

"We don't know what the challenges will be and that's C++'s greatest strength."

It's a really big world "out there" and the increase in the number of users of one language does not imply the decrease in the numbers of another. I suspect we have reached the point where if you can count your users to the nearest million, you don't count as a major language. Similarly, I got into a discussion with some friends about how many billions of lines of C++ code there were "out there." We concluded that we did not know, but it didn't require much thought to demonstrate that the plural was appropriate.

One simple thing that confuses many discussions of language use/popularity is the distinction between relative and absolute measures. For example, I say that C++ use is growing when I see user population grow by 200,000 programmers from 3.1M to 3.3M. However, somebody else may claim that "C++ is dying" because it's "popularity" has dropped from 16 percent to 10 percent of the total number of users. Both claims could be simultaneously true as the number of programmers continues to grow and especially as what is considered to be programming continues to change. I think that C++ is more than holding its own in its traditional core domains, such as infrastructure, systems programming, embedded systems, and applications with serious time and/or space constraints. It never was dominant for scripting web applications.

Most of the popularity measures seem to measure buzz/noise, which is basically counting mentions on the web. That's potentially very misleading. Ten people learning a scripting language will make much more noise than a thousand full time programmers using C++, especially if the thousand C++ programmers are working on a project crucial to a company—such programmers typically don't post and are often not allowed to. My worry is that such measures may actually measure the number of novices and thus be an indication of a worsening shortage of C++ programmers. Worse, managers and academics may incautiously take such figures seriously (as a measure of quality) and become part of a vicious circle.

Q What are the biggest challenges that the software industry will face five, 10, or maybe 20 years from now? What makes C++ the right choice for software developers who are facing these challenges?

A We don't know what the challenges will be and that's C++'s greatest strength. C++ is more general, flexible, and efficient (in run time and space) than alternatives and will therefore be a leading candidate to address these unknown challenges.

That relates to the technical challenges. The major problems for C++ are social. The C++ community is far less well organized than other communities; we don't have an owner, a dictator, a community web site, a journal, or a conference that defines the language. The closest we get to a focal point in the C++ community is the ISO standards committee and it's in itself a diverse body. However, as a focus for the C++ community, the committee is not ideal. For example, language and tools implementers are heavily overrepresented compared to the user community. Similarly, I don't think that the educational establishment really has a grip on C++. Too many don't understand it, partly because they don't understand the needs of the industries that are the primary users of C++. We need better, tutorials, courses, textbooks, web sites, etc. We also need better ways of evaluating and distributing libraries and tools. There are so many libraries out there of such varying quality that many users give up and say "C++ doesn't have X" even though there is a choice of a dozen reasonable Xs available.

It is often said that the past is not a good predictor of the future, but it's the only predictor we have, so I can recommend my HOPL paper "Evolving a language for and in the real world: C++ 1991-2006." It discusses the background for the current use of C++, the work on C++0x, and tries to describe C++ relative to the problems it faces.

Danny Kalev is a certified system analyst and software engineer specializing in C++ and theoretical linguistics. He has an MA degree in general linguistics and is the author of Informit C++ Reference Guide and The ANSI/ISO Professional C++ Programmer's Handbook. Danny was a member of the C++ standards committee between 1997 and 2000. Danny recently finished his MA in general linguistics summa cum laude. In his spare time he likes to listen to classical music, read Victorian literature, and explore new natural and formal languages alike. He also gives lectures about programming and applied linguistics at academic institutes.