Al Stevens Interviews Alex Stepanov

This interview appeared in the March 1995 issue of *Dr. Dobb's Journal*, and is reprinted with permission.

Tell us something about your long-term interest in generic programming.

I started thinking about generic programming in the late 70s when I observed that some algorithms depended not on some particular implementation of a data structure but only on a few fundamental semantic properties of the structure. I started going through many different algorithms, and I found that most algorithms can be abstracted away from a particular implementation in such a way that efficiency is not lost. Efficiency is a fundamental concern of mine. It is silly to abstract an algorithm in such a way that when you instantiate it back it becomes inefficient.

At that time I thought that the right way of doing this kind of research was to develop a programming language, which is what I started doing with two of my friends, Deepak Kapur, who at present is a professor at State University of New York, Albany, and David Musser, professor at Rensselaer Polytechnic Institute. At that time the three of us worked at the General Electric Research Center at Schenectady, NY. We started working on a language called Tecton, which would allow people to describe algorithms associated with what we called generic structures, which is just a collection of formal types and properties of these types. Sort of mathematical stuff. We realized that one can define an algebra of operations on these structures, you can refine them, you can enrich them, and do all sorts of things.

There were some interesting ideas, but the research didn't lead to practical results because Tecton was functional. We believed Backus's idea that we should liberate programming from the von Neumann style, and we didn't want to have side effects. That limited our ability to handle very many algorithms that require the notion of state and side effects.

The interesting thing about Tecton, which I realized sometime in the late 70s, was that there was a fundamental limitation in the accepted notion of an abstract data type. People usually viewed abstract data types as something which tells you only about the behavior of an object and the implementation is totally hidden. It was commonly assumed that the complexity of an operation is part of implementation and that abstraction ignores complexity. One of the things that is central to generic programming as I understand it now, is that complexity, or at least some general notion of complexity, has to be associated with an operation.

Let's take an example. Consider an abstract data type stack. It's not enough to have Push and Pop connected with the axiom wherein you push something onto the stack and after you pop the stack you get the same thing back. It is of paramount importance that pushing the stack is a constant time operation regardless of the size of the stack. If I implement the stack so that every time I push it becomes slower and slower, no one will want to use this stack.

We need to separate the implementation from the interface but not at the cost of totally ignoring complexity. Complexity has to be and is a part of the unwritten contract between the module and its user. The reason for introducing the notion of abstract data types was to allow interchangeable software modules. You cannot have interchangeable modules unless these modules share similar complexity behavior. If I replace one module with another module with the same functional behavior but with different complexity tradeoffs, the user of this code will be unpleasantly surprised. I could tell him anything I like about data abstraction, and he still would not want to use the code. Complexity assertions have to be part of the interface.

Around 1983 I moved from GE Research to the faculty of the Polytechnic University, formerly known as Brooklyn Polytechnic, in NY. I started working on graph algorithms. My principal collaborator was Aaron Kershenbaum, now at IBM Yorktown Heights. He was an expert in graph and network algorithms, and I convinced him that some of the ideas of high order and generic programming were applicable to graph algorithms. He had some grants and provided me with support to start working with him to apply these ideas to real network algorithms. He was interested in building a toolbox of high order generic components so that some of these algorithms could be implemented, because some of the network algorithms are so complex that while they are theoretically analyzed, but never implemented. I decided to use a dialect of Lisp called Scheme to build such a toolbox. Aaron and I developed a large library of components in Scheme demonstrating all kinds of programming techniques. Network algorithms were the primary target. Later Dave Musser, who was still at GE Research, joined us, and we developed even more components, a fairly large library. The library was used at the university by graduate students, but was never used commercially. I realized during this activity that side effects are important, because you cannot really do graph operations without side effects. You cannot replicate a graph every time you want to modify a vertex. Therefore, the insight at that time was that you can combine high order techniques when building generic algorithms with disciplined use of side effects. Side effects are not necessarily bad; they are bad only when they are misused.

In the summer of 1985 I was invited back to GE Research to teach a course on high order programming. I demonstrated how you can construct complex algorithms using this technique. One of the people who attended was Art Chen, then the manager of the Information Systems Laboratory. He was sufficiently impressed to ask me if I could produce an industrial strength library using these techniques in Ada, provided that I would get support. Being a poor assistant professor, I said yes, even though I didn't know any Ada at the time. I collaborated with Dave Musser in building this Ada library. It was an important undertaking, because switching from a dynamically typed language, such as Scheme, to a strongly typed language, such as Ada, allowed me to realize the importance of strong typing. Everybody realizes that strong typing helps in catching errors. I discovered that strong typing, in the context of Ada generics, was also an instrument of capturing designs. It was not just a tool to catch bugs. It was also a tool to think. That work led to the idea of orthogonal decomposition of a component space. I realized that software components belong to different categories. Object-oriented programming aficionados think that everything is an object. When I was working on the Ada generic library, I realized that this wasn't so. There are things that are objects. Things that have state and change their state are objects. And then there are things that are not objects. A binary search is not an object. It is an algorithm. Moreover, I realized that by decomposing the component space into several orthogonal dimensions, we can reduce the number of components, and, more importantly, we can provide a conceptual framework of how to design things.

Then I was offered a job at Bell Laboratories working in the C++ group on C++ libraries. They asked me whether I could do it in C++. Of course, I didn't know C++ and, of course, I said I could. But I couldn't do it in C++, because in 1987 C++ didn't have templates, which are essential for enabling this style of programming. Inheritance was the only mechanism to obtain genericity and it was not sufficient.

Even now C++ inheritance is not of much use for generic programming. Let's discuss why. Many people have attempted to use inheritance to implement data structures and container classes. As we know now, there were few if any successful attempts. C++ inheritance, and the programming style associated with it are dramatically limited. It is impossible to implement a design which includes as trivial a thing as equality using it. If you start with a base class X at the root of your hierarchy and define a virtual equality operator on this class which takes an argument of the type X, then derive class Y from class X. What is the interface of the equality? It has equality which compares Y with X. Using animals as an example (OO people love animals), define mammal and derive giraffe from mammal. Then define a member function mate, where animal mates with animal and returns an animal. Then you derive giraffe from animal and, of course, it has a function mate where giraffe mates with animal and returns an animal. It's definitely not what you want. While mating may not be very important for C++ programmers, equality is. I do not know a single algorithm where equality of some kind is not used.

You need templates to deal with such problems. You can have template class animal which has member function mate which takes animal and returns animal. When you instantiate giraffe, mate will do the right thing. The template is a more powerful mechanism in that respect.

However, I was able to build a rather large library of algorithms, which later became part of the Unix System Laboratory Standard Component Library. I learned a lot at Bell Labs by talking to people like Andy Koenig and Bjarne Stroustrup about programming. I realized that C/C++ is an important programming language with some fundamental paradigms that cannot be ignored. In particular I learned that pointers are very good. I don't mean dangling pointers. I don't mean pointers to the stack. But I mean that the general notion of pointer is a powerful tool. The notion of address is universally used. It is incorrectly believed that pointers make our thinking sequential. That is not so. Without some kind of address we cannot describe any parallel algorithm. If you attempt to describe an addition of n numbers in parallel, you cannot do it unless you can talk about the first number being added to the second number, while the third number is added to the fourth number. You need some kind of indexing. You need some kind of address to describe any kind of algorithm, sequential or parallel. The notion of an address or a location is fundamental in our conceptualizing computational processes---algorithms.

Let's consider now why C is a great language. It is commonly believed that C is a hack which was successful because Unix was written in it. I disagree. Over a long period of time computer architectures evolved, not because of some clever people figuring how to evolve architectures---as a matter of fact, clever people were pushing tagged architectures during that period of time---but because of the demands of different programmers to solve real problems. Computers that were able to deal just with numbers evolved into computers with byte-addressable memory, flat address spaces, and pointers. This was a natural evolution reflecting the growing set of problems that people were solving. C, reflecting the genius of Dennis Ritchie, provided a minimal model of the computer that had evolved over 30 years. C was not a quick hack. As computers evolved to handle all kinds of problems, C, being the minimal model of such a computer, became a very powerful language to solve all kinds of problems in different domains very effectively. This is the secret of C's portability: it is the best representation of an abstract computer that we have. Of course, the abstraction is done over the set of real computers, not some imaginary computational devices. Moreover, people could understand the machine model behind C. It is much easier for an average engineer to understand the machine model behind C than the machine model behind Ada or even Scheme. C succeeded because it was doing the right thing, not because of AT&T promoting it or Unix being written with it.

C++ is successful because instead of trying to come up with some machine model invented by just contemplating one's navel, Bjarne started with C and tried to evolve C further, allowing more general programming techniques but within the framework of this machine model. The machine model of C is very simple. You have the memory where things reside. You have pointers to the consecutive elements of the memory. It's very easy to understand. C++ keeps this model, but makes things that reside in the memory more extensive than in the C machine, because C has a limited set of data types. It has structures that allow a sort of an extensible type system, but it does not allow you to define operations on structures. This limits the extensibility of the type system. C++ moved C's machine model much further toward a truly extensible type system.

In 1988 I moved to HP Labs where I was hired to work on generic libraries. For several years, instead of doing that I worked on disk drives, which was exciting but was totally orthogonal to this area of research. I returned to generic library development in 1992 when Bill Worley, who was my lab director established an algorithms project with me being its manager. C++ had templates by then. I discovered that Bjarne had done a marvelous job at designing templates. I had participated in several discussions early on at Bell Labs about designing templates and argued rather violently with Bjarne that he should make C++ templates as close to Ada generics as possible. I think that I argued so violently that he decided against that. I realized the importance of having template functions in C++ and not just template classes, as some people believed. I thought, however, that template functions should work like Ada generics, that is, that they should be explicitly instantiated. Bjarne did not listen to me and he designed a template function mechanism where templates are instantiated implicitly using an overloading mechanism. This particular technique became crucial for my work because I discovered that it allowed me to do many things that were not possible in Ada. I view this particular design by Bjarne as a marvelous piece of work and I'm very happy that he didn't follow my advice.

When did you first conceive of the STL and what was its original purpose?

In 1992, when the project was formed, there were eight people in it. Gradually the group diminished, eventually becoming two people, me and Meng Lee. While Meng was new to the area---she was doing compilers for most of her professional life---she accepted the overall vision of generic programming research, and believed that it could lead to changing software development at the point when very few people shared this belief. I do not think that I would be able to build STL without her help. (After all, STL stands for Stepanov and Lee...) We wrote a huge library, a lot of code with a lot of data structures and algorithms, function

objects, adaptors, and so on. There was a lot of code, but no documentation. Our work was viewed as a research project with the goal of demonstrating that you can have algorithms defined as generically as possible and still extremely efficient. We spent a lot of time taking measurements, and we found that we can make these algorithms as generic as they can be, and still be as efficient as hand-written code. There is no performance penalty for this style of programming! The library was growing, but it wasn't clear where it was heading as a project. It took several fortunate events to lead it toward STL.

When and why did you decide to propose STL as part of the ANSI/ISO Standard C++ definition?

During the summer of 1993, Andy Koenig came to teach a C++ course at Stanford. I showed him some of our stuff, and I think he was genuinely excited about it. He arranged an invitation for me to give a talk at the November meeting of the ANSI/ISO C++ Standards Committee in San Jose. I gave a talk entitled "The Science of C++ Programming." The talk was rather theoretical. The main point was that there are fundamental laws connecting basic operations on elements of C++ which have to be obeyed. I showed a set of laws that connect very primitive operations such as constructors, assignment, and equality. C++ as a language does not impose any constraints. You can define your equality operator to do multiplication. But equality should be equality, and it should be a reflexive operation. A should be equal to A. It should be symmetric. If A is equal to B, then B should be equal to A. And it should be transitive. Standard mathematical axioms. Equality is essential for other operations. There are axioms that connect constructors and equality. If you construct an object with a copy constructor out of another object, the two objects should be equal. C++ does not mandate this, but this is one of the fundamental laws that we must obey. Assignment has to create equal objects. So I presented a bunch of axioms that connected these basic operations. I talked a little bit about axioms of iterators and showed some of the generic algorithms working on iterators. It was a two-hour talk and, I thought, rather dry. However, it was very well received. I didn't think at that time about using this thing as a part of the standard because it was commonly perceived that this was some kind of advanced programming technique which would not be used in the "real world". I thought there was no interest at all in any of this work by practical people.

I gave this talk in November, and I didn't think about ANSI at all until January. On January 6 I got a mail message from Andy Koenig, who is the project editor of the standard document, saying that if I wanted to make my library a part of the standard, I should submit a proposal by January 25. My answer was, "Andy, are you crazy?" to which he answered, "Well, yes I am crazy, but why not try it?"

At that point there was a lot of code but there was no documentation, much less a formal proposal. Meng and I spent 80-hour weeks to come up with a proposal in time for the mailing deadline. During that time the only person who knew it was coming was Andy. He was the only supporter and he did help a lot during this period. We sent the proposal out, and waited. While doing the proposal we defined a lot of things. When you write things down, especially when you propose them as a standard, you discover all kinds of flaws with your design. We had to re-implement every single piece of code in the library, several hundred components, between the January mailing and the next meeting in March in San Diego. Then we had to revise the proposal, because while writing the code, we discovered many flaws.

Can you characterize the discussions and debate in the committee following the proposal? Was there immediate support? Opposition?

We did not believe that anything would come out of it. I gave a talk, which was very well received. There were a lot of objections, most of which took this form: this is a huge proposal, it's way too late, a resolution had been passed at the previous meeting not to accept any major proposals, and here is this enormous thing, the largest proposal ever, with a lot of totally new things. The vote was taken, and, interestingly enough, an overwhelming majority voted to review the proposal at the next meeting and put it to a vote at the next meeting in Waterloo, Ontario.

Bjarne Stroustrup became a strong supporter of STL. A lot of people helped with suggestions, modifications, and revisions. Bjarne came here for a week to work with us. Andy helped constantly. C++ is a complex language, so it is not always clear what a given construct means. Almost daily I called Andy or Bjarne to ask whether such-and-such was doable in C++. I should give Andy special credit. He conceived of STL as part of the standard library. Bjarne became the main pusher of STL on the committee. There were other people who were helpful: Mike Vilot, the head of the library group, Nathan Myers of Rogue Wave, Larry Podmolik of Andersen Consulting. There were many others.

The STL as we proposed it in San Diego was written in present C++. We were asked to rewrite it using the new ANSI/ISO language features, some of which are not implemented. There was an enormous demand on Bjarne's and Andy's time trying to verify that we were using these non-implemented features correctly.

People wanted containers independent of the memory model, which was somewhat excessive because the language doesn't include memory models. People wanted the library to provide some mechanism for abstracting memory models. Earlier versions of STL assumed that the size of the container is expressible as an integer of type `size_t` and that the distance between two iterators is of type `ptrdiff_t`. And now we were told, why don't you abstract from that? It's a tall order because the language does not abstract from that; C and C++ arrays are not parameterized by these types. We invented a mechanism called "allocator," which encapsulates information about the memory model. That caused grave consequences for every component in the library. You might wonder what memory models have to do with algorithms or the container interfaces. If you cannot use things like `size_t`, you also cannot use things like `T*` because of different pointer types (`T*`, `T huge *`, etc.). Then you cannot use references because with different memory models you have different reference types. There were tremendous ramifications on the library.

The second major thing was to extend our original set of data structures with associative data structures. That was easier, but coming up with a standard is always hard because we needed something which people would use for years to come for their containers. STL has from the point of view of containers, a very clean dichotomy. It provides two fundamental kinds of container classes: sequences and associative containers. They are like regular memory and content-addressable memory. It has a clean semantics explaining what these containers do.

When I arrived at Waterloo, Bjarne spent a lot of time explaining to me that I shouldn't be concerned, that most likely it was going to fail, but that we did our best, we tried, and we should be brave. The level of expectation was low. We expected major opposition. There was some opposition but it was minor. When the vote was taken in Waterloo, it was totally surprising because it was maybe 80% in favor and 20% against. Everybody expected a battle, everybody expected controversy. There was a battle, but the vote was overwhelming.

What effect does STL have on the class libraries published in the ANSI/ISO February 1994 working paper?

STL was incorporated into the working paper in Waterloo. The STL document is split apart, and put in different places of the library parts of the working paper. Mile Vilot is responsible for doing that. I do not take active part in the editorial activities. I am not a member of the committee but every time an STL–related change is proposed, it is run by me. The committee is very considerate.

Several template changes have been accepted by the committee. Which ones have impact on STL?

Prior to the acceptance of STL there were two changes that were used by the revised STL. One is the ability to have template member functions. STL uses them extensively to allow you to construct any kind of a container from any other kind of a container. There is a single constructor that allows you to construct vectors out of lists or out of other containers. There is a templatized constructor which is templatized on the iterator, so if you give a pair of iterators to a container constructor, the container is constructed out of the elements which are specified by this range. A range is a set of elements specified by a pair of iterators, generalized pointers, or addresses. The second significant new feature used in STL was template arguments which are templates themselves, and that's how allocators, as originally proposed, were done.

Did the requirements of STL influence any of the proposed template changes?

In Valley Forge, Bjarne proposed a significant addition to templates called "partial specialization," which would allow many of the algorithms and classes to be much more efficient and which would address a problem of code size. I worked with Bjarne on the proposal and it was driven by the need of making STL even more efficient. Let me explain what partial specialization is. At present you can have a template function parameterized by class T called `swap(T&, T&)` and swaps them. This is the most generic possible `swap`. If you want to specialize `swap` and do something different for a particular type, you can have a function `swap(int&, int&)`, and which does integer swapping in some different way. However it was not possible to have an intermediate partial specialization, that is, to provide a template function of the following form:

```
template <class T> void swap(vector<T>&, vector<T>&);
```

This form provides a special way to swap vectors. This is an important problem from an efficiency point of view. If you swap vectors with the most generic `swap`, which uses three assignments, vectors are copied three times, which takes linear time. However, if we have this partial specialization of `swap` for vectors that swap two vectors, then you can have a fast, constant time operation, that moves a couple of pointers in the vector headers. That would allow `sort`, for example, to work on vectors of vectors much faster. With the present STL, without partial specialization, the only way to make it work faster is for any particular kind of `vector`, such as `vector<int>`, to define its own `swap`, which can be done but which puts a burden on the programmer. In very many cases, partial specialization would allow algorithms to be more effective on some generic classes. You can have the most generic `swap`, a less generic `swap`, an even less generic `swap`, and a totally specific `swap`. You can do partial specialization, and the compiler will find the closest match. Another example is `copy`. At present the `copy` algorithm just goes through a sequence of elements defined by iterators and copies them one by one. However, with partial specialization we can define a template function:

```
template <class T> T ** copy(T**,T**,T**);
```

This will efficiently copy a range of pointers by using `memcpy`, because when we're copying pointers we don't have to worry about construction and destruction and we can just move bits with `memcpy`. That can be done once and for all in the library and the user doesn't need to be concerned. We can have particular specializations of algorithms for some of the types. That was a very important change, and as far as I know it was favorably received in Valley Forge and will be part of the Standard.

What kinds of applications beyond the standard class libraries are best served by STL ?

I have hopes that STL will introduce a style of programming called generic programming. I believe that this style is applicable to any kind of application, that is, trying to write algorithms and data structures in the most generic way. Specifying families or categories of such structures satisfying common semantic requirements is a universal paradigm applicable to anything. It will take a long time before the technology is understood and developed. STL is a starting point for this type of programming.

Eventually we will have standard catalogs of generic components with well–defined interfaces, with well–defined complexities. Programmers will stop programming at the micro level. You will never need to write a binary search routine again. Even now, STL provides several binary search algorithms written in the most generic way. Anything that is binary–searchable can be searched by those algorithms. The minimum requirements that the algorithm assumes are the only requirements that the code uses. I hope that the same thing will happen for all software components. We will have standard catalogs and people will stop writing these things.

That was Doug McIlroy's dream when he published a famous paper talking about component factories in 1969. STL is an example of the programming technology which will enable such component factories. Of course, a major effort is needed, not just research effort, but industrial effort to provide programmers with such catalogs, to have tools which will allow people to find the components they need, and to glue the components together, and to verify that their complexity assumptions are met.

STL does not implement a persistent object container model. The `map` and `multimap` containers are particularly good candidates for persistent storage containers as inverted indexes into persistent object databases. Have you done any work in

that direction or can you comment an such implementations?

This point was noticed by many people. STL does not implement persistence for a good reason. STL is as large as was conceivable at that time. I don't think that any larger set of components would have passed through the standards committee. But persistence is something that several people thought about. During the design of STL and especially during the design of the allocator component, Bjarne observed that allocators, which encapsulate memory models, could be used to encapsulate a persistent memory model. The insight was Bjarne's, and it is an important and interesting insight. Several object database companies are looking at that. In October 1994 I attended a meeting of the Object Database Management Group. I gave a talk on STL, and there was strong interest there to make the containers within their emerging interface to conform to STL. They were not looking at the allocators as such. Some of the members of the Group are, however, investigating whether allocators can be used to implement persistency. I expect that there will be persistent object stores with STL–conforming interfaces fitting into the STL framework within the next year.

`Set`, `multiset`, `map`, and `multimap` are implemented with a red–black tree data structure. Have you experimented with other structures such as B*trees?

I don't think that would be quite right for in–memory data structures, but this is something that needs to be done. The same interfaces defined by STL need to be implemented with other data structures–––skip lists, splay trees, half–balanced trees, and so on. It's a major research activity that needs to be done because STL provides us with a framework where we can compare the performance of these structures. The interface is fixed. The basic complexity requirements are fixed. Now we can have meaningful experiments comparing different data structures to each other. There were a lot of people from the data structure community coming up with all kinds of data structures for that kind of interface. I hope that they would implement them as generic data structures within the STL framework.

Are compiler vendors working with you to implement STL into their products?

Yes. I get a lot of calls from compiler vendors. Pete Becker of Borland was extremely helpful. He helped by writing some code so that we could implement allocators for all the memory models of Borland compilers. Symantec is going to release an STL implementation for their Macintosh compiler. Edison Design Group has been very helpful. We have had a lot of support from most compiler vendors.

STL includes templates that support memory models of 16–bit MS–DOS compilers. With the current emphasis on 32–bit, flat model compilers and operating systems, do you think that the memory–model orientation will continue to be valid?

Irrespective of Intel architecture, memory model is an object, which encapsulates the information about what is a pointer, what are the integer size and difference types associated with this pointer, what is the reference type associated with this pointer, and so on. Abstracting that is important if we introduce other kinds of memory such as persistent memory, shared memory, and so on. A nice feature of STL is that the only place that mentions the machine–related types in STL–––something that refers to real pointer, real reference–––is encapsulated within roughly 16 lines of code. Everything else, all the containers, all the algorithms, are built abstractly without mentioning anything which relates to the machine. From the point of view of portability, all the machine–specific things which relate to the notion of address, pointer, and so on, are encapsulated within a tiny, well–understood mechanism. Allocators, however, are not essential to STL, not as essential as the decomposition of fundamental data structures and algorithms.

The ANSI/ISO C Standards committee treated platform–specific issues such as memory models as implementation details and did not attempt to codify them. Will the C++ committee be taking a different view of these issues? If so, why?

I think that STL is ahead of the C++ standard from the point of view of memory models. But there is a significant difference between C and C++. C++ has constructors and operator new, which deal with memory model and which are part of the language. It might be important now to look at generalizing things like operator new to be able to take allocators the way STL containers take allocators. It is not as important now as it was before STL was accepted, because STL data structures will eliminate the majority of the needs for using new. Most people should not allocate arrays because STL does an effective job in doing so. I never need to use new in my code, and I pay great attention to efficiency. The code tends to be more efficient than if I were to use new. With the acceptance of STL, new will sort of fade away. STL also solves the problem of deleting because, for example, in the case of a vector, the destructor will destroy it on the exit from the block. You don't need to worry about releasing the storage as you do when you use new. STL can dramatically minimize the demand for garbage collection. Disciplined use of containers allows you to do whatever you need to do without automatic memory management. The STL constructors and destructors do allocation properly.

The C++ Standard Library subcommittee is defining standard namespaces and conventions for exception handling. Will STL classes have namespaces and throw exceptions?

Yes they will. Members of the committee are dealing with that, and they are doing a great job.

How different from the current STL definition will the eventual standard definition be? Will the committee influence changes or is the design under tighter control?

It seems to be a consensus that there should not be any major changes to STL.

How can programmers gain an early experience with STL in anticipation of it becoming a standard?

They can download the STL header files from `butler.hpl.hp.com` under `/stl` and use it with Borland or IBM compiler, or with any other compiler powerful enough to handle STL The only way to learn some style of programming is by programming. They need to look at examples and write programs in this style.

You are collaborating with P.J. (Bill) Plauger to write a book about STL. What will be the emphasis of the book and when is it scheduled to be published?

It is scheduled to be published in the summer of 1995 and is going to be an annotated STL implementation. It will be similar to Bill's books on the Standard C Library and the Draft Standard C++ Library. He is taking the lead on the book, which will serve as a standard reference document on the use of the STL. I hope to write a paper with Bjarne that will address language/library interactions in the context of C++/STL. It might lead to another book.

A lot more work needs to be done. For STL to become a success, people should do research experimenting with this style of programming. More books and articles need to be written explaining how to program in this style. Courses need to be developed. Tutorials need to be written. Tools need to be built which help people navigate through libraries. STL is a framework and it would be nice to have a tool with which to browse through this framework.

What is the relationship between generic programming and object-oriented programming?

In one sense, generic programming is a natural continuation of the fundamental ideas of object-oriented programming---separating the interface and implementation and polymorphic behavior of the components. However, there is a radical difference. Object-oriented programming emphasizes the syntax of linguistic elements of the program construction. You have to use inheritance, you have to use classes, you have to use objects, objects send messages. Generic programming does not start with the notion of whether you use inheritance or you don't use inheritance. It starts with an attempt to classify or produce a taxonomy of what kinds of things are there and how they behave. That is, what does it mean that two things are equal? What is the right way to define equality? Not just actions of equality. You can analyze equality deeper and discover that there is a generic notion of equality wherein two objects are equal if their parts, or at least their essential parts are equal. We can have a generic recipe for an equality operation. We can discuss what kinds of objects there are. There are sequences. There are operations on sequences. What are the semantics of these operations? What types of sequences from the point of view of complexity tradeoffs should we offer the user? What kinds of algorithms are there on sequences? What kind of different sorting functions do we need? And only after we develop that, after we have the conceptual taxonomy of the components, do we address the issue of how to implement them. Do we use templates? Do we use inheritance? Do we use macros? What kind of language technology do we use? The fundamental idea of generic programming is to classify abstract software components and their behavior and come up with a standard taxonomy. The starting point is with real, efficient algorithms and data structures and not with the language. Of course, it is always embodied in the language. You cannot have generic programming outside of a language. STL is done in C++. You could implement it in Ada. You could implement it in other languages. They would be slightly different, but there are some fundamental things that would be there. Binary search has to be everywhere. Sort has to be everywhere. That's what people do. There will be some modification on the semantics of the containers, slight modifications imposed by the language. In some languages you can use inheritance more, in some languages you have to use templates. But the fundamental difference is precisely that generic programming starts with semantics and semantic decomposition. For example, we decide that we need a component called `swap`. Then we figure out how this particular component will work in different languages. The emphasis is on the semantics and semantic classification, while object-orientedness, especially as it has evolved, places a much stronger emphasis, and, I think, too much of an emphasis, on precisely how to develop things, that is, using class hierarchies. OOP tells you how to build class hierarchies, but it doesn't tell you what should be inside those class hierarchies.

What do you see as the future of STL and generic programming?

I mentioned before the dream of programmers having standard repositories of abstract components with interfaces that are well understood and that conform to common paradigms. To do that there needs to be a lot more effort to develop the scientific underpinnings of this style of programming. STL starts it to some degree by classifying the semantics of some fundamental components. We need to work more on that. The goal is to transform software engineering from a craft to an engineering discipline. It needs a taxonomy of fundamental concepts and some laws that govern those concepts, which are well understood, which can be taught, which every programmer knows even if he cannot state them correctly. Many people know arithmetic even if they never heard of commutativity. Everybody who graduated from high school knows that 2+5 is equal to 5+2. Not all of them know that it is a commutative property of addition. I hope that most programmers will learn the fundamental semantic properties of fundamental operations. What does assignment mean? What does equality mean? How to construct data structures.

At present C++ is the best vehicle for this style of programming. I have tried different languages and I think that C++ allows this marvelous combination of abstractness and efficiency. However, I think that it is possible to design a language based on C and on many of the insights that C++ brought into the world, a language which is more suitable to this style of programming, which lacks some of the deficiencies of C++, in particular its enormous size. STL deals with things called concepts. What is an iterator? Not a class. Not a type. It is a concept. (Or, if we want to be more formal, it is what Bourbaki calls a structure type, what logicians call a theory, or what type theory people call a sort.) It is something which doesn't have a linguistic incarnation in C++. But it could. You could have a language where you could talk about concepts, refine them, and then finally form them in a very programmatic kind of way into classes. (There are, of course, languages that deal with sorts, but they are not of much use if you want to sort.) We could have a language where we could define something called forward iterator, which is just defined as a concept in STL---it doesn't have a C++ incarnation. Then we can refine forward iterator into bidirectional iterator. Then random iterator can be refined from that. It is possible to design a language which would enable even far greater ease for this style of programming. I am fully convinced that it has to be as efficient and as close to the machine as are C and C++. And I do believe that it is possible to construct a language that allows close approximation to the machine on the one hand and has the ability to deal with very abstract entities on the other hand. I think that abstractness can be even greater than it is in C++ without creating a gap between underlying machines. I think that generic programming can influence language research and that we will have practical languages, which are easy to use and are well suited for that style of programming. From that you can deduce what I am planning to work on next.

STL Main Page

Contact Us | Site Map | Trademarks | Privacy | Using this site means you accept its Terms of Use