An Interview with A. Stepanov

by *Graziano Lo Russo*

Edizioni Infomedia srl

Question:
May you introduce yourself?

Answer:
I was born in Moscow, USSR, on November 16, 1950, and studied Mathematics at the Moscow State University. But I never became a mathematician. I could not really get excited about Tamagawa numbers, Coxeter groups and other things that I was supposed to specialize in. Hardy's hope that his mathematics is never going to be applied is not for me. I need to do something a little bit more real. I was fortunate, however, to see some great mathematicians at work and became totally immune to a pseudo-mathematical rigor that unfortunately is so common in Computer Science. So becoming a programmer was a really good thing for me. In 1972 I became a member of a team developing a new minicomputer to be used to control large hydroelectric power stations. I participated in all parts of the design, from architecture and hardware testing to OS (my first published paper was on real-time operating systems) and programming tools. I learned first hand about both software reliability - power stations are hard to reboot - and efficiency - the water is coming down in real time.

At that time I also discovered books of two great computer scientists from whose work I learned the scientific foundation of my trade: Donald Knuth and Edsger Dijkstra. Knuth taught me the answers. Dijkstra taught me the questions. Time and time again I come back to their works for new insights. My next important career step was spending 5 years at the Computer Science Branch of General Electric Research Center in Schenectady, NY. I worked on a very high level language called Tecton and read a lot: from a plethora of papers on programming language design to Logical Summa of William of Occam - Aristotle and medieval logicians knew a lot about different kinds of logical structures that appear in the natural languages and their formal properties. At that time I started a fruitful research collaboration with Dave Musser which is still going on. In 1984 I became an assistant professor at Polytechnic University in Brooklyn, NY. Teaching Computer Science did me a lot of good - I managed to teach all kinds of graduate courses, learning a lot of new material in the process. I also developed a large library of algorithms and data structures in Scheme. This work led to a development (together with Dave Musser) of Ada Generic Library. After a brief stint at Bell Labs, where I worked on a library of algorithms in C++, I moved to HP Labs in Palo Alto (1988). I spent the next 4 years working on storage systems: I had to learn how to program disk controllers. In 1993 I was given a brief opportunity to return to my research on generic programming. STL is a result of this. In 1995 I moved on to Silicon Graphics, where I have been trying to establish a group to work on further development of STL.

Question:
You mentioned William of Occam. William of Occam used to say "Entia non sunt multiplicanda" - I could translate that as "[abstract] objects are unnecessary". It seems that you have applied the razor of Occam to OOP. Starting from algorithms rather than

from objects kinds of reminds me the medieval quarrel on universals. Is it right?

Answer:
It is a pretty analogy, but I do not think it is right. I never thought of OO as related to a realist philosophy and I am not a nominalist at all. As a matter of fact, the Franciscan school: Alexander of Hales, Bonaventure and Scotus are much closer to Augustinian/Platonist tradition. Occam was a strange fellow anyways. As the editor of his Opera Omnia Gideon Gal used to say: "But the fellow was really mad!"

Question:
For most Italian readers, the name "Stepanov" goes in pair with STL. Does STL mean Standard Template Library or Stepanov and Lee? And what was the role of D.Musser and A.Koenig in STL history?

Answer:
Well, it does really mean Standard Template Library. I did make a joke in my interview with Dr. Dobb's journal about STL standing for "Stepanov and Lee," but it was a joke. I have been collaborating with Dave Musser for almost 20 years. Our collaboration has been so close that it is difficult to say who contributed what. His is not on the list of the official STL authors only because in the short period of time when the proposal for the standard had to been written, he was busy doing something else. Andy Koenig is responsible for explaining to me the structure of the abstract C machine. STL, in a sense, is an application of generic programming techniques that Dave Musser and I have been developing to a C machine model. If it were not for Andy, I would be still dealing with boxed, heap-allocated objects and pining for garbage collection. And, of course, Andy and Bjarne Stroustrup are responsible for putting STL into the standard. Meng Lee was a perfect collaborator at the stage when it was necessary to move from beautiful ideas to a complete implementation. She kept me focused - I tend to lose interest after the problem is solved - if I know the solution, why bother making it known to the rest of the world. She was putting grueling hours into the code and the document. In a sense, she was the only person who believed that something practical could come out of the stuff; I think, that at that point both Dave Musser and I actually lost hope that we could explain to anybody what we had been doing for quite a while.

Question:
What is the origin of STL? Has STL been conceived to be what it is now, that is "the" C++ Standard Library, or does it come from some other project? Could you tell us a history of STL?

Answer:
In 1976, still back in the USSR, I got a very serious case of food poisoning from eating raw fish. While in the hospital, in the state of delirium, I suddenly realized that the ability to add numbers in parallel depends on the fact that addition is associative. (So, putting it simply, STL is the result of a bacterial infection.) In other words, I realized that a parallel reduction algorithm is associated with a semigroup structure type. That is the fundamental point: algorithms are defined on algebraic structures. It took me another couple of years to realize that you have to extend the notion of structure by adding complexity requirements to regular axioms. And than it took 15 years to make it work. (I am still not sure that I have been successful in getting the point across to anybody outside the small circle of my friends.) I believe that iterator theories are as central to Computer Science as theories of rings or Banach spaces are central to

Mathematics. Every time I would look at an algorithm I would try to find a structure on which it is defined. So what I wanted to do was to describe algorithms generically. That's what I like to do. I can spend a month working on a well known algorithm trying to find its generic representation. So far, I have been singularly unsuccessful in explaining to people that this is an important activity. But, somehow, the result of the activity - STL - became quite successful.

Question:
I used to think at complexity requirements in STL as mere constraints on the efficiency of the implementation. You seem to imply that complexity is a true functional requirement. Is it so?

Answer:
You select different algorithms depending on complexity of the fundamental operations provided by the data structure. Disk and tape are functionally equivalent as SCSI storage devices, but woe to the software designer who would attempt to use the tape as if it were a disk. In terms of STL - you can always implement p + n for forward iterators, so why bother providing random access iterators?

Question:
What about Generic Programming? I only found some columns by A.Koenig in JOOP on "generic programming". Generic programming is definitively absent in most C++ programming books, including Coplien, Meyer, Stroustrup, Lippman and so on. I think STL could be described as "Programming C++ the way you would never thought possible". Do you agree?

Answer:
STL, at least for me, represents the only way programming is possible. It is, indeed, quite different from C++ programming as it was presented and still is presented in most textbooks. But, you see, I was not trying to program in C++, I was trying to find the right way to deal with software. I have been searching for a language in which I could express what I wanted to say for a long time. In other words, I know what I want to say. I can say it in C++, I can say it in Ada, I can say it in Scheme. I adapt myself to the language, but the essence of what I am trying to say is language independent. So far, C++ is the best language I've discovered to say what I want to say. It is not the ideal medium, but I can do more in it than in any other language I tried. It is actually my hope that someday there will be a language designed specifically with generic programming in mind.

Question:
Could you explain to a modest C++ programmer what Generic Programming is, what is the relation of Generic Programming with C++ and STL, and how did you come to use Generic Programming in a C++ context?

Answer:
Generic programming is a programming method that is based in finding the most abstract representations of efficient algorithms. That is, you start with an algorithm and find the most general set of requirements that allows it to perform and to perform efficiently. The amazing thing is that many different algorithms need the same set of requirements and there are multiple implementations of these requirements. The

analogous fact in mathematics is that many different theorems depend on the same set of axioms and there are many different models of the same axioms. Abstraction works! Generic programming assumes that there are some fundamental laws that govern the behavior of software components and that it is possible to design interoperable modules based on these laws. It is also possible to use the laws to guide our software design. STL is an example of generic programming. C++ is a language in which I was able to produce a convincing example.

Question:
I think STL and Generic Programming mark a definite departure from the common C++ programming style, which I find is almost completely derived from SmallTalk. Do you agree?

Answer:
Yes. STL is not object oriented. I think that object orientedness is almost as much of a hoax as Artificial Intelligence. I have yet to see an interesting piece of code that comes from these OO people. In a sense, I am unfair to AI: I learned a lot of stuff from the MIT AI Lab crowd, they have done some really fundamental work: Bill Gosper's Hakmem is one of the best things for a programmer to read. AI might not have had a serious foundation, but it produced Gosper and Stallman (Emacs), Moses (Macsyma) and Sussman (Scheme, together with Guy Steele). I find OOP technically unsound. It attempts to decompose the world in terms of interfaces that vary on a single type. To deal with the real problems you need multisorted algebras - families of interfaces that span multiple types. I find OOP philosophically unsound. It claims that everything is an object. Even if it is true it is not very interesting - saying that everything is an object is saying nothing at all. I find OOP methodologically wrong. It starts with classes. It is as if mathematicians would start with axioms. You do not start with axioms - you start with proofs. Only when you have found a bunch of related proofs, can you come up with axioms. You end with axioms. The same thing is true in programming: you have to start with interesting algorithms. Only when you understand them well, can you come up with an interface that will let them work.

Question:
Can I summarize your thinking as "find the [generic] data structure inside an algorithm" instead of "find the [virtual] algorithms inside an object"?

Answer:
Yes. Always start with algorithms.

Question:
This mean a radical change of mind from both imperative and OO thinking. What are the benefits, and the drawbacks, of this paradigm compared to the "standard" OO programming of SmallTalk or, say, Java?

Answer:
My approach works, theirs does not work. Try to implement a simple thing in the object oriented way, say, `max`. I do not know how it can be done. Using generic programming I can write:

```
template <class StrictWeakOrdered>
inline StrictWeakOrdered& max(StrictWeakOrdered& x,
StrictWeakOrdered& y) {
return x < y ? y : x;
```

```
}

and
template <class StrictWeakOrdered>
inline const StrictWeakOrdered& max(const StrictWeakOrdered& x,
const StrictWeakOrdered& y) {
return x < y ? y : x;
}
```

(you do need both & and const &). And then I define what strict weak ordered means. Try doing it in Java. You can't write a generic `max()` in Java that takes two arguments of some type and has a return value of that same type. Inheritance and interfaces don't help. And if they cannot implement max or swap or linear search, what chances do they have to implement really complex stuff? These are my litmus tests: if a language allows me to implement max and swap and linear search generically - then it has some potential.

Question:
Java is a very new language, still it lacks templates, so it prevents using Generic Programming. Everything must be a class. What do you think of Java?

Answer:
I spent several months programming in Java. Contrary to its authors prediction, it did not grow on me. I did not find any new insights - for the first time in my life programming in a new language did not bring me new insights. It keeps all the stuff that I never use in C++ - inheritance, virtuals - OO gook - and removes the stuff that I find useful. It might be successful - after all, MS DOS was - and it might be a profitable thing for all your readers to learn Java, but it has no intellectual value whatsoever. Look at their implementation of hash tables. Look at the sorting routines that come with their "cool" sorting applet. Try to use AWT. The best way to judge a language is to look at the code written by its proponents. "Radix enim omnium malorum est cupiditas" - and Java is clearly an example of a money oriented programming (MOP). As the chief proponent of Java at SGI told me: "Alex, you have to go where the money is." But I do not particularly want to go where the money is - it usually does not smell nice there.

Question:
Do you think template-based programming and Generic Programming will be adopted by the majority of C++ programmers, or will they be confined to STL, somewhat like manipulators, which were never used outside the iostream library?

Answer:
I do not know. It will take a long time before the ideas behind STL enter the mainstream. We will know in about 10-15 years if anything comes out of all this.

Question:
One thing has always amazed me, is how quickly STL was adopted by the C++ standardization committees. I mean, these committees are known to be very cautious and conservative. How do you explain that?

Answer:
The support of Bjarne Stroustrup was crucial. Bjarne really wanted STL in the standard and if Bjarne wants something, he gets it. He is as stubborn as a mule. He even forced me to make changes in STL that I would never make for anybody else - I am also stubborn, but he is the most single minded person I know. He gets things done. It took

him a while to understand what STL was all about, but when he did, he was prepared to push it through. He also contributed to STL by standing up for the view that more than one way of programming was valid - against no end of flak and hype for more than a decade, and pursuing a combination of flexibility, efficiency, overloading, and type-safety in templates that made STL possible. I would like to state quite clearly that Bjarne is the preeminent language designer of my generation.

Question:
STL is full of creative uses of templates, such as symbolic types exported from classes, or the pattern matching of a set of overloaded algorithms onto iterator tags. Surely enough, no standard C++ Programming book speaks about those idioms. How did you come to these C++ code idioms?

Answer:
I knew exactly what I was trying to accomplish. So I tweaked the language until I was able to get by. But it took me many years to discover all the techniques. And I had many false starts. For example, I spent years trying to find some use for inheritance and virtuals, before I understood why that mechanism was fundamentally flawed and should not be used. I am very happy that nobody could see all the intermediate steps - most of them were very silly. It takes me years to come up with anything half decent. It also helped that Bjarne was willing to put certain features in the language just to enable some of my idioms. He once refered to it as "just in time language design."

Question:
What do you think is the best way to teach Generic Programming and STL to C++ programmers? Do you think inheritance and other OO techniques should be learnt before, after or at the same time as STL?

Answer:
I plan to teach a course on Generic Programming at SGI. I do hope that it will lead to a book, but I am a notoriously lazy writer - I never finish papers unless I have a collaborator who does.

Question:
I have done a search on Lycos for your papers and I only found two titles: the STL manual and a resume of you presentation of STL to the standardization committee.

Answer:
Well, I am lazy, but not that lazy. I probably published 20 papers and a book. Many of them are on different STL sites. (Dave Musser's site probably has several.)

Question:
Which book?

Answer:
The book is "The Ada Generic Library: Linear List Processing Packages", by David R. Musser and Alexander A. Stepanov, Compass Series, Springer-Verlag, 1989. It is not really worth reading.

Question:
STL pushes C++ compilers to their limits. Contemporary C++ compilers are still unable to correctly compile some STL code. How could you develop and test STL?

Answer:
I do have a lot of gray hair as a result of trying to compile STL. The unfortunate reality is that a lot of code in the present implementation of STL is suboptimal because of the compiler limitations and bugs of the compilers I had to use when I was developing STL. Fortunately, I had help from Bjarne in figuring out what certain unimplemented features are supposed to do. It does help a lot if you can ask the language designer what a given construct really does.

Question:
How did allocators come into STL? What do you think of them?

Answer:
I invented allocators to deal with Intel's memory architecture. They are not such a bad ideas in theory - having a layer that encapsulates all memory stuff: pointers, references, ptrdiff_t, size_t. Unfortunately they cannot work in practice. For example,

```
vector<int, alloc1> a(...);
vector<int, alloc2> b(...);
```

you cannot now say:

```
find(a.begin(), a.end(), b[1]);
```

`b[1]` returns a `alloc2::reference` and not `int&`. It could be a type mismatch. It is necessary to change the way that the core language deals with references to make allocators really useful.

Question:
Could this point out a serious drawback of C++ templates? I can customize a class using a template argument, but different specializations are not type compatible. Instead, different subclasses (in OO sense) of a class are type compatible with the root class.

Answer:
I think that the problem is deeper than that. `T*` is hardwired into the language. In general, I believe that it is necessary to design a programming language from the ground up to enable generic programming in a consistent way. I wish that somebody would hire me to do just that.

Question:
I find two hash table implementations in the D.Musser site, and they were both working and quite smart - much smarter than hash tables commonly found in class libraries. Why were hash tables not included into STL?

Answer:
Politics. They have to be in. Our new implementation of STL does contain them. In general, we need to develop a mechanism for adding things to STL. After all, STL is an extensible framework, it begs to be extended. There are many data structures that are missing, for example, singly linked lists, matrices, and graphs. SGI is willing to lead the way in extending STL.

Question:
Are you still working on STL? Doing what?

Answer:
My group at SGI, - Matt Austern, Hans Boehm and myself, - just finished a new version of STL. It includes hash containers, thread safe memory allocation and spectacular web documentation. SGI released it into the public domain (http://www.sgi.com/Technology/STL). We hope that we will be able to keep STL growing. Multidimensional data structures, persistence, multithreading are among the things that we plan do add. Well, it is not clear how long we can keep doing this stuff. There are no support from my management for any of the STL/generic programming activities. It is very hard to explain to them why SGI should pay for extensions. (It was equaly hard to explain it to HP management. They canceled my project 5 months after STL was accepted into the standard.)

Question:
It seems that STL has still some drawbacks. First, HP STL is not thread safe. Then, templates cause a lot of code to be put into header files. You cannot have true template libraries, nor DLLs or shared libraries of (uninstantiated) templates: STL is mostly a compile- time library. The last drawback is that almost no CASE tool and no OOD methodology effectively supports Generic Programming. For instance, no CASE tool lets you define generic functions. Also, only the Booch notation is somewhat able to express templates, but resulting diagrams are not intuitive (to me, at least).

Answer:
SGI STL is thread safe. The acceptance of separate compilation into the standard - designed by my SGI collegues John Wilkinson, Jim Dehnert and Matt Austern solves your second problem. It was voted into the standard, but it will take a while before there are compilers that can do separate compilation. I do believe that separate compilation would eventually require shipping shared libraries of templates. That was the main reason I initiated SGI work on separate compilation. It is not that difficult to design tools that will deal with Generic Programming. And I am quite sure that if there is some market for it, Grady Booch will change his notation to handle generic programming.

Question:
One painful thing for the C++ programmer is that standardization committees seem to be unaware of each other. OMG has just defined a standard for distributed programming (CORBA), but the mapping of CORBA to C++ is unaware of STL. They define their own set of classes, such as Sequence<T> and CORBA::String. The same problem arises for ODMG and its ODMG-93 standard for Object Databases. Why does this happen? Are things going to change?

Answer:
I am old enough to remember all the networking standards that were coming out in the seventies. Who remembers them now?

Question:
What could Generic Programming be in a distributed environment? Generic Programming is based on the idea that the compiler "knows" all possible types at compile time. This is not realistic in a distributed environment. Should we think at a kind of ORB integrated with a compiler? Or is Generic Programming simply not suited to Distributed Programming - in that case, Java would be right?

Answer:
Generic Programming has nothing to do with run time vs. compile time. The problem that I find with OOP is not just that it is slow, but that it does not allow me to express simplest possible algorithms. Again, the signature of max is:

```
max: T x T -> T
```

It is not expressible in Java, because the inheritance from some class or interface `T` changes it into:

```
max: T' x T -> T
```

You need covariant signature transformation and an ability to obtain types from types, a notion of a virtual type if you like, a v-table containing type descriptors.

Question:
What is an iterator?

Answer:
An iterator is a union of two theories. The first theory is a theory of name (handle, cookie, address). A name is something that points to something else. (`operator*`). We call it Trivial Iterator theory in our site. In addition to the dereferencing, it has equality defined that satisfies the following axiom:

```
i == j iff &*i == &*j
```

That is, two iterators are equal if and only if they point to the same object. (Equality, of course, needs to satisfy all the standard axioms.) The second theory is the theory of successor operation `(++i)` with its refinements: successor - predcessor (`++` and `--`) and addition (`++`, `--`, `+` and `-`) with the standard axioms. And, of course, pointer is just a model of random access iterator.

Question:
Some argue that a pointer is a way to hack any conceivable horrible and weird thing into memory. Java and Delphi made it right when they disallowed pointers.

Answer:
Disallowing pointers that allow you to do pointer arithmetic is a good thing. Pointer arithmetic should be allowed only where it is really allowed in C and C++, namely, for pointers into arrays. But disallowing pointers in general is a very silly thing. You cannot get generic swap unless you have pointers or references in the language.

Question:
"category" is an overloaded word in the C++ community. How do you call iterator categories?

Answer:

I often call these things concepts.

Question:
I have tried to develop a STL-like singly linked list. But I chose not to implement the size() member function, because I did not want the overhead of keeping a counter just to implement size() in a constant time, as stated by the C++ CD. Was it the right decision?

Answer:
size() used to be linear time in case of STL lists. It was the right decision since if a user wants to keep a count it is easy to do, but usually you do not need it and it makes splice linear time (it used to be constant). But the standard committee insisted that I change it to constant time. I had no choice. We will have to change this requirement eventually.

Question:
I am studying how to express a tree into STL and I have some problems: every node has one father but two sons. Moving to the father could be represented as operator--, but I would need two different operator++ to move to the sons. How can iterators deal with not linear structures, such as trees o graphs?

Answer:
Even on a sequence you have different iterators. Reverse iterators are just one example. Stride iterators are very important and will have to be put into STL eventually.

Question:
I must confess my ignorance. What are stride iterators?

Answer:
Go from `i` to `i+5` to `i+10`.

Question:
What's the difference with random iterators?

Answer:
Stride iterator is an iterator adaptor that takes a random access iterator range and provides a (random access iterator) such that `++` on it goes through a stride (a sequence of iterator n steps apart).

Question:
How does this relate with the problem of traversing a tree?

Answer:
I did not mean to say that stride iterators or reverse iterators have anything to do with tree traversal, but that there could be multiple iterator types on a data structure for different iteration orders - in case of trees pre, in and post order traversals.

Question:
A frequent dilemma for me was: should I design this function as a member function or as a generic (global) function? what has been the rationale of this decision in STL?

Answer:
Make it global if it at all possible. It would be much nicer if begin and end were global - it would allow us to define them for C arrays. It would be so much nicer if `operator*` was global with the default definitions:

```
template <class T>
T& operator*(T& x) { return x;}

template <class T>
const T& operator*(const T& x) { return x;}
```

It would allow us to write:

```
copy(0, 25, ostream_iterator<int>("\n"));
```

In general, for non-iterator objects `operator*` should return the object itself, the "meaning" of a non-naming thing is a thing itself. I would even love to write constructors and destructors as global functions. You could do some amazing stuff if this is allowed in the language.

Question:
If I define a generic `operator*` your way, and in my program I define:

```
template <class T> class SmartPtr {
  T* ptr;
 public:
  SmartPtr(T* _ptr = 0) : ptr(_ptr){}
  T* operator*() const {return ptr;}
  // ...
};
```

than shouldn't I get ambiguity in this code:

```
int i=0;
SmartPtr<int> sp(&i);
int j= *i; // apply SmartPtr<int>::operator* or
// operator<SmartPtr<int>? both are user defined
```

Answer:
No, you do not. Your definition matches better (the unification is deeper) than the global one. That is what partial specialization is all about.

Question:
Now, a bunch of curiosities of mine: why does not STL have something like a "sorted container" adaptor?

Answer:
Set is a sorted container.

Question:
Set is not an adaptor. Why was it possible to write a heap, but not a sorted container, out of any container supporting random access iterators?

Answer:
Remember that it was very hard to push through STL because of its size. I had to

throw away dosens of useful components. (Think what happened to hash tables.)

Question:
Why is the function `__adjust_heap` in `<heap.h>` not documented ? This function is necessary to use the heap in the Dijkstra algorithm.

Answer:
I had great difficulty pushing heap functions into the standard. Originally, I wanted all auxiliary functions in STL to be visible, but it was not politically possible.

Question:
It is hard for me to figure something political about a heap functions.

Answer:
It was not any particular function, but the number of them. Bjarne is personally responsible for reducing the number of components in STL by a factor of two. He was trying to make it as small as possible to placate the opposition.

Question:
Have your ever been in Italy, for business or for pleasure?

Answer:
Yes, I spent 10 days in Pisa, visited Florence and Lucca. I dream of going to Assisi - I am a Franciscan at heart. I cry during the second act of Tosca and the third act of La Traviata. I keep Dante (in Italian!) on my bed stand. I love pasta, prosciutto and Chianti, - while I am a traditionalist, I look much more like John XXIII than Pius XII.

Question:
You must know Italian very well if you can read Dante in its original version: very few Italians could!

Answer:
Well, my Italian is very poor. The way I do read Dante in Italian is by having an English translation in front of me. I read a stanza aloud in Italian and then read the translation.

Graziano: Thank you very much, Alex. I hope you will be able to come to Italy and see Assisi soon, reading Dante in the old town of Assisi - maybe the most charming medieval town in Italy .

Graziano Lo Russo has a degree in Electronic Engineering from the Politecnico of Turin (Italy) and is a specialist in object orientated techniques. He is a contributing editor to many Italian programming magazines. He can be reached at lorusso@infomedia.it.