Foreword

When Dave Musser asked me to write an extended foreword to the second edition of this book, I jumped on the opportunity. First, Dave is my closest professional friend; we have been collaborating for over 20 years, and without Dave there would be no STL. So honoring his request is by itself a privilege. It also gives me an opportunity to say a few words about what I had in mind while designing STL.

To use a tool, it is useful to understand not just the instructions for using it but also the principles that guided its designers. The main goal of this foreword is to present you with the principles behind STL. I'll conclude with some musings.

STL was designed with four fundamental ideas in mind:

- Abstractness
- Efficiency
- Von Neumann computational model
- Value semantics

Abstractness. Some of you might have heard that STL is an example of a programming technique called "generic programming." This is so. Some of you might have also heard that generic programming is a style of programming using C++ templates. This is not so. Generic programming has nothing to do with C++ or templates. Generic programming is a discipline that studies systematic organization of abstract software components. Its objective is to develop a taxonomy of algorithms, data structures, memory allocation mechanisms, and other software artifacts in a way that allows the highest level of reuse, modularity, and usability.

To allow the greatest degree of usability, one has to try to analyze all possible extensions. For example, when a famous computer scientist saw my version of Euclid's algorithm for finding the greatest common divisor of two quantities,

```
template <typename T> T gcd(T m, T n) {
  while (n != 0) {
    T t = m % n;
    m = n;
    n = t;
  }
  return m;
}
```

he objected that the algorithm is not correct since it returns −1 when called with 1 and −1 as its arguments, and therefore the common divisor returned is not the greatest. He suggested that I fix the problem by changing the last line to

```
return m < 0 ? -m : m;
```

Unfortunately, if you do this the algorithm will not work for many important extensions: polynomials, Gaussian integers, and so on. It would require the set of elements on which we operate to be totally ordered. The problem disappears if we use a more abstract (and algorithmically more meaningful) definition of greatest common divisor: a divisor that is divisible by any other divisor. That definition allows for nonunique solutions: in the case of integers both 6 and −6 are greatest common divisors of 24 and 30. This actually corresponds to what mathematicians have been doing for the last several hundred years.

The classification of software components should deal only with useful components. It would be ridiculous to introduce a concept of semisequence—a sequence that has multiple beginnings but only one end—since we do not know any data structures that look like that nor any algorithms that could operate on them.

After we organize things systematically, we can ensure the consistency of their interfaces. That is, interfaces to two components should be the same to the same degree that the behavior of the components is the same. That allows us to implement algorithms that work on multiple components—generic algorithms. It also makes it possible to use the library. If a programmer masters STL's `vector`, it is not going to be too hard to learn to use STL's `list` and even easier to learn to use `deque`. It is my belief that the interfaces that allow the greatest possible degree of abstract programming are also the interfaces that are easiest to learn. (This presupposes that a person is learning things from scratch. It is hard to convince a hardened Lisp programmer that comparing with the past-the-end iterator is a better way than testing for `nil`.)

In many respects the ideas of generic programming are very similar to the ideas of abstract algebra. Those of you who took a course dealing with groups, rings, and fields should be able to see where classification of iterators is coming from.[1]

As mathematics organizes theorems around different abstract theories, generic programming organizes algorithms around different abstract concepts. So the task of the library designer is to find all interesting algorithms, find the minimal requirements that allow these algorithms to work, and organize them around these requirements. In general requirements are described through a set of acceptable expressions and their semantics. For example, STL does not state that `++` on an iterator must be defined as a member function of a class. It just states that if `i` is an iterator and if it can be dereferenced, then `++i` is a valid expression.

Efficiency. While mathematics often deals with objects that cannot be constructed at all or could be constructed only if given an arbitrarily long time, computer science makes efficiency an explicit concern. It is not enough to know that an operation can be done. It is important to know that it will be done reasonably fast. To assure that, STL does several things.

First, it makes complexity requirements a part of each interface. When concepts such as iterators are specified, certain complexity requirements are given. A programmer can be certain that doing `++` on an iterator does not depend dramatically on where in the sequence it is. Dereferencing should be equally fast—it is not legal to implement list iterators with a structure

[1]In general, I believe that mathematical culture is essential for a good software engineer. Sadly enough, nowadays one goes through college—and through graduate school—without any exposure to real mathematics. I would urge all of you to keep reading mathematics throughout your career. There are some remarkable books out there—I highly recommend the following three books by John Stillwell: *Numbers and Geometry*, *Mathematics and Its History*, and *Elements of Algebra*; after you are done with them, consider *Geometry: Euclid and Beyond* by Robin Hartshorne and *Visual Complex Analysis* by Tristan Needham.

containing the pointer to the list's head and the integer index. (It should be noted that while the operational semantics of the operations can be specified rigorously by specifying the set of valid expressions and their semantics, the complexity is specified informally; a totally new insight is needed to find a way for specifying complexity requirements in a rigorous but practically useful way.)

Second, STL takes great care not to hide any part of a data structure that allows efficient access. Instead of providing get and put methods for operating on a container—the favorite method of textbook writers—the pointer to the value is exposed so that fields could be modified in place. One can write

```
i->second = 5;
```

instead of

```
pair<int, int> tmp = my_vector.get(i);
tmp.second = 5;
my_vector.put(i, tmp);
```

The fact that iterators to elements of a vector do not survive the periodic reallocations is noted, and it is assumed that STL users can learn to deal with it by either preallocating enough storage or storing indices and not iterators.

Great care was taken to see that all the generic algorithms in STL are state of the art and as efficient as hand-coded ones (being quite precise, that they are as efficient as hand-coded ones when a good optimizing compiler—such as Kuck and Associates' C++ compiler—is used).

Von Neumann Computational Model. Although abstract mathematics uses simple numeric facts as its basis of abstraction—one should not forget that mathematics is an experimental science—what should we use as our basis of abstraction to come up with a generic abstract framework? It is my firm belief that the only solid basis is the architectures of real computers. It is important to remember that modern computer architectures are a result of many years of evolution guided by the need to solve more and more diverse problems. Byte-addressable memory and pointers are not the artifacts we inherited from some archaic hardware designs—archaic hardware designs did not have bytes and there were no pointers; one wrote loops with the help of self-modifying code—but the results of architecture catching up with the needs of applications.[2] If we are interested in designing a generic framework for numerical types, it is important to understand the working of built-in numeric types, not just the mathematical theory of integers and real numbers.

The most important new concept in computer science that was not already present in mathematics is the concept of address. Making addresses, not just values, a part of our computational model was the revolutionary step that enabled all the progress from 72 addresses in the Mark I to millions of Internet addresses. In many respects, the most controversial part of STL is the fact that it makes addresses and their conceptual classification the cornerstone of the whole edifice. (This statement might appear strange to a practical programmer, but the academic community has spent decades trying to eliminate addresses altogether in doing what is called "functional programming.") In mathematical terms, the idea underlying STL is that different data structures correspond to different address algebras, different ways of connecting addresses together. A set of operations that move from one address in the data structure to the next corresponds to iterators. A set of operations that add and delete addresses to and from the data structure corresponds to containers.

While the STL classification of iterators (input, output, forward, bidirectional, random access) is sufficient for all the fundamental sequence algorithms, further categories of iterators need to be defined for STL to be properly extended to deal with multidimensional structures. (As a matter of fact, even for many fundamental sequence algorithms, two-dimensional iterators are needed to speed them up in cases of (1) nonuniform accesses as, for example, is the case with deque iterators or cache lines and (2) multiprocessor implementations.)

Value Semantics. STL views containers as a generalization of structures. As the structure owns its components, so does a container own its components. When you copy structures, all their components are copied. When

[2]It is very important for a good programmer to understand what really goes under the hood of a high-level programming language. It is important to know at least a couple of different architectures well. Since I recommended a bunch of mathematical books, let me also suggest a couple of computer books: John Hennessy and David Patterson's *Computer Architecture: A Quantitative Approach* is, in my opinion, the most important computer science book; one gets, however, a wonderful additional perspective if it is supplemented with *Computer Architecture: Concepts and Evolution*, by Gerrit Blaauw and Fred Brooks, especially the second part of the book, "The Computer Zoo," which covers some remarkable historical designs.

a structure is destroyed, all its components are destroyed. The same happens with containers. These properties are essential features that allow structures and containers to model the key attribute of real-life things—the relationship between whole and part. Of course, the whole-part relationship is not the only kind of relationship in the real world, and the rest of the relationships need to be modeled with iterators.[3] It is my belief that the confusion between a part and a relation, which is so common in object-oriented languages and libraries, is a major source of conceptual confusion in the modeling of the real world as well as the main reason that they absolutely require garbage collection. STL is not object-oriented—not only in the way it uses global generic algorithms, but more significantly, in the fact that it separates the notions of having an object as a part and pointing to an object. It assumes that

```
T a = b;
```

creates a copy of an object, with all parts being distinct, not just another pointer to the same object. Specifications of those algorithms in STL that use assignment (`sort`, `partition`, `remove`, and so on) require this value semantics. In the STL universe objects never share parts (unless, of course, one object is a part of the other).

In general, STL assumes that for any type on which it operates the semantics of copy constructors, destructors, assignment, and equality and their relations are the same as for built-in types. In addition, STL assumes that for those objects for which operators `<`, `>`, `<=`, and `>=` are defined, their semantics is the same as for built-in types, or, mathematically speaking, they define a total ordering. (One of the gripes I have against C++ is that C++ does not require the semantics of fundamental operations to be consistent with the semantics of built-in types; one can define an operator `=` to do multiplication. Operator overloading is good only if used in a highly disciplined way; otherwise, it can cause great harm.)

Musings. STL was not designed to be a part of the C++ Standard Library. It was designed to be the first library of generic algorithms and data structures. It so happened that C++ was the only language in which I could implement such a library to my personal satisfaction. In the five

[3]For example, while my leg is my part, my lawyer is not. If I am destroyed, my leg is destroyed; if I am copied, my leg is copied. My lawyer is another human being, and while my death might affect him in various ways—like a lot of pointers to a dead client, known as dangling pointers—he is not going to be automatically destroyed.

years since STL has been widely available, many people have made claims that they can do STL-like things in their favorite language: Ada-95, ML, Dylan, Eiffel, Java, and so on. Maybe they can. As far as I can see, they have not. I wish they could. I wish someone would construct a language more suitable to generic programming than C++. After all, one gets by in C++ by the skin of one's teeth. Fundamental concepts of STL, things like iterators and containers, are not describable in C++ since STL depends on rigorous sets of requirements that do not have any linguistic representation in C++. (They are, of course, defined in the standard, but they are defined in English.)

The whole point of STL is that it is an extensible framework. While STL is widely used, my hopes for the creation of many libraries of generic components have not been fulfilled. As far as I can determine the reason that such libraries are not created is that there are no financial mechanisms for supporting the work. One cannot make money out of fundamental algorithms. They have to be designed for the entire industry by small teams of component craftsmen. While I have been lucky, on a couple of occasions, to receive funding from large computer companies to do STL work, it cannot be done in a serious way until some reliable way of funding the work is found. It is my hope that the U.S. government or alternatively the EU will fund a small but effective organization dedicated to producing generic software components. And I mean not research but actual production of well-organized, documented, generic, and efficient components. Please write to your elected representatives.

STL presupposes a very different way of teaching computer science. What 99 percent of programmers need to know is not how to build components but how to use them. STL presupposes a different way of running software organizations. People who write their own code, instead of using standard components, should be dealt with like people who propose designing nonstandard, proprietary CPUs. Can we ever move software into the industrial age? I wonder . . .

Alexander Stepanov
January 2001

Foreword to the First Edition

What is STL? STL, or the Standard Template Library, is a general-purpose library of generic algorithms and data structures. It makes a programmer more productive in two ways: first, it contains a lot of different components that can be plugged together and used in an application, and more importantly, it provides a framework into which different programming problems can be decomposed.

The framework defined by STL is quite simple: two of its most fundamental dimensions are algorithms and data structures. The reason that data structures and algorithms work together seamlessly is, paradoxically enough, the fact that they do not know anything about each other. Algorithms are written in terms of iterator categories: abstract data-accessing methods. To enable different algorithms to work in terms of these conceptual categories, STL establishes rigid rules that govern the behavior of iterators. For example, if any two iterators are equal then the results of dereferencing them must be equal. It is only because in STL all such rules are stated explicitly that it is possible to write code that knows nothing about a particular implementation of a data structure.

While it is my experience that using STL can dramatically improve programming productivity, such an improvement is possible only if a programmer is fully cognizant of the structure of the library and is familiar with a style of programming that it advocates. How can a programmer learn this style? The only way is to use it and extend it. To do this, however, one needs a place to start. This book is such a place.

The authors bring special qualifications to the writing of this book. Dave Musser has been doing research that led to STL for over fifteen years. Quoting from the original STL manual: "Dave Musser . . . contributed to all aspects of the STL work: design of the overall structure, semantic requirements, algorithm design, complexity analysis, and performance measurements." Atul Saini was the first person to recognize the commercial potential of STL and committed his company to selling its production version even before it was accepted by the C++ standards committee.

I hope that this book's publication will help programmers enjoy using STL as much as I do.

Alexander Stepanov
October 1995