Algorithm Concepts for Standard Libraries

October 4, 2001

1 Point of View

My point of view in this talk: that of writers of the official standard for a software library, who strive to make the library components

- easy to understand and use properly by applications programmers;
- implementable using algorithms and data structures that are correct, robust, and efficient;
- yet, implementable in different ways by different compiler/library vendors in order to take advantage of special characteristics of the platform(s) on which they are marketing the library. (By a "platform" I mean a particular hardware/OS/compiler combination.).

Overall, these goals contribute to acceptance of standard library components as reliable building blocks for constructing application programs that can be ported from one platform to another without change.

The ISO/ANSI C++ standard (1997) has these goals but only partially achieves them.

2 The C++ (language and) library standard

- Developed over an eight year period, this standard includes detailed requirements for what components must be in any C++ library, and what kinds of services they must provide.
- Included among the components are fundamental data processing algorithms and data structures (the part known as the Standard Template Library, or STL).
- Each STL component specification describes:
  - interface (how to use it)
  - functionality (what it does)
  - *performance guarantees* (how efficient is it)

3 Nature of the performance standards

In writing performance standards, the library standard writer has the same goals as previously stated for the overall library, including that the components be implementable in different ways by different compiler/library vendors in order to take advantage of special characteristics of their platform(s).

- Ordinarily such tailoring involves making tradeoffs in resource use.
- Even on the same hardware, differences in underlying software such as different operating systems could be a factor.

4 How strictly should performance guarantees be expressed?

The problem we address here is, how can a library standard be written so that

- library implementors have the freedom to take some advantage of hardware platform characteristics, yet
- application programmers have sufficient guarantees about the performance of library components that they can easily port their programs from one platform to another.

5 Performance guarantees in the C++ standard

- The requirements are mostly stated in traditional O-notation.
- In some cases exact or approximate bounds are given for operation counts, of some principal operation (like comparison operations, in sorting algorithm descriptions).

6 A typical algorithm specification in the C++ standard

heapsort (Simplified from **partial_sort**'s description.)

Prototype:

```
template<class RandomAccessIterator>
void heapsort(RandomAccessIterator first,
              RandomAccessIterator last)
```

Effects: Sorts the elements in the range [first, last), in place.

Complexity: It takes approximately $N \log N$ comparisons, where $N = \text{last} - \text{first}$.

7 A better specification of heapsort

See **partial_sort**'s web page in the Silicon Graphics STL documentation (http://www.sgi.com/tech/stl).

8 Can we do better?

More precise requirements are needed. Here is a view expressed in a newsgroup discussion of C++ standard library performance requirements:

> *I've yet to be convinced that placing requirements on computational complexity really accomplished anything useful. . . . I guess in theory this was done to guarantee that all programs would be efficient. In reality, it doesn't go far enough to guarantee such a thing, and I can't think of a way of rewriting the specification so it would either.*

9 Some traditional approaches

9.1 Actual running times

Start with measurements of actual running times on a particular platform.

- Insights gained are of course not necessarily portable to other platforms.
- Multi-platform experimentation is expensive, and can result in proliferation of data that is hard to manage.
- It may be difficult to generalize from it, and thus to use it as a basis for setting standards.

9.2 Instruction cycle and memory access counts

A slightly more abstract approach is to measure machine instruction cycles and memory accesses for a given machine architecture.

- Often easier to measure instruction counts than actual times:
  - Some machines provide a hardware register that keeps count.
  - Machine simulators can be used, with the instruction cycle count as one of the simulation results reported.
- Still too machine dependent; the effort required to derive closed formulas for the bounds is not justified by the limited portability of the results.

9.3 Instruction cycles on an abstract machine

Still more abstract: program and study the algorithm for a hypothetical architecture designed to resemble many common hardware architectures:

- E.g., Don Knuth's hardware level analyses in TAOCP in terms of programs for his MIX and MMIX machine specifications, which allow study of trade-offs that simply cannot be seen or properly assessed from more abstract levels like asymptotic analysis.
- The performance results become somewhat less accurate when extrapolated to a real architecture.
- But in many cases they still form a reasonable basis for choosing between different algorithms or algorithm variants.

- Because of the wider applicability of the results, investment in deriving analytical bounds is more justifiable and useful. [TAOCP, especially]

For characterizing performance of algorithm components in standard libraries, however, we still haven't reached the level of abstraction that is needed. This is certainly the case when the library provides *generic components*.

10 The challenge of generic software components

- Libraries like STL (and MTL, BGL, ...) provide generic components, which are designed with parameters representing infinite sets of abstractions (such as sets of types, which are represented in C++ for example by template parameters).
- For such generic components, machine instruction counts or memory accesses cannot be measured or derived analytically unless the component's parameters are all instantiated with specific types to produce a nongeneric instance.

- But each generic component has infinitely many such instances.

  The best we could do is try to measure or analyze the machine level statistics and catalog them for the "most common cases."

- Yet, while genericity seems to complicate matters, in fact it also contributes new ideas about how to express algorithm performance.

11 How generic programming can contribute to solving our problem

- In brief outline, the approach I suggest is:
  - Develop *algorithm concept hierarchies* similar to previously developed hierarchies for container and iterator concepts.
  - Use these algorithm concepts to present and organize performance requirements for a standard library's algorithm components.
  - Start with a way of expressing these requirements that is well matched to the level of abstraction of generic algorithms.
  - Extend it so that it takes into account key hardware characteristics such as cache size and speed.

12 Programming with concepts

Generic programming can be viewed as "programming with concepts," as illustrated in the following diagrams.

Programming with Concepts

13 Formally, what is a "concept"?

Definition A *concept* is a pair of sets (R, A), where R is a set of requirements and A is a set of abstractions, such that an abstraction belongs to A if and only if it satisfies all the requirements in R. [R. Wille].

- Concept refinement: more requirements (means fewer abstractions in the concept but each is more specialized).
- Refinement relation yields a *concept lattice* or *hierarchy*.
- More extended notion: *concept web*: a concept hierarchy that includes cross-references to other useful information related to the concepts (examples, experiment reports, animations, etc.)

- *The Tecton Concept Library* describes a concept hierarchy that formally captures many algebraic concepts (SemiGroup, Monoid, Group, Ring, Field, etc.) [R. Loos, D. Musser, S. Schupp, C. Schwarzweller].
- The Silicon Graphics STL documentation [M. Austern] is a concept web with extensive development of Container, Iterator, and Functor concepts.
- Yet there has been little development of classifications and documentation of *algorithms* in concept hierarchies or concept webs.

14 A sample algorithm concept hierarchy

- The following diagram depicts an overall (sequential) algorithm concept hierarchy, with detailed development of one subconcept,
  - sequence algorithms, like those in STL [based in part on pre-STL ideas jointly developed with Alex Stepanov (unpublished)]

15 Another heapsort specification: as it might appear in an algorithm concept hierarchy

Prototype (same)

Refinement of: Sequence Sorting Algorithm (§A.11), therefore of Comparison Based (§A.6), Permuting (§A.10), Sequence Algorithm (§A.5).

Effects: Standard effects of a Sequence Sorting Algorithm (§A.11). In brief: the elements in [first, last) after execution are a permutation of the original elements in the range, and they are in nondecreasing order according the comparison operator.

Asymptoptic complexity: Let $N = \text{last} - \text{first}$.

- Average case (random data): $O(N \log N)$
- Worst case: $O(N \log N)$

16 Heapsort specification, continued

Complexity in terms of operation counts:

- Average case (random data (§C)):
  Value comparisons: $N \log_2 N + 0.36N$
  Value assignments: $1.2N \log_2 N + 3.2N$
  Iterator operations: $12.4N \log_2 N + 10N$
  Integer operations: $14.5N \log_2 N + 17N$
- Worst case: ...
- See also Table 1 for sample counts on random data for heapsort and other sorting algorithms available in the library.
- For any implementation the corresponding principal operation counts must be bounded by the above formulas multiplied by the Standard Tolerance, $(1 + \epsilon)$.

17 In general, what is the role of principal operation counts?

- An algorithm concept is expressed in terms of certain other concepts (like Random Access Iterator, Comparison, etc.)
  - Each of those other concepts requires certain operations, which we call *principal operations*, to exist.
  - Therefore, express the performance requirements for the algorithm concept in terms of counts of principal operations.
- Present measured and (where possible) analytically-derived bounds.
- Performance requirements are then expressed in terms of these bounds times a constant factor.

18 How are the bounds determined and used?

- Library standard writers measure or analyze operation counts for several variants of an algorithm and express required bounds (i.e., choose the "fudge factor" $(1 + \epsilon)$) so that the desired variants are allowed.
- Library implementors still have some freedom to choose a different variant of the algorithm in order to take advantage of platform characteristics. (But not, typically, to choose an entirely different algorithm.)
- Application programmers can rely on each of the algorithm implementations they use on platforms X, Y, . . . to operate within the stated bounds.

19 Why principal operation counting is not enough

- Looking only at principal operation counts ignores important hardware differences, in

  available instruction sets, number and speed of arithmetic units, registers, size and speed of caches and memory, etc.

- This abstractness can lead to suboptimal choices of algorithms for a particular task.

20 Extending principal operation counting

- How we might extend principal operation counting to take better account of hardware differences:
  - Express algorithms with additional parameters that capture key hardware characteristics as a concept, e.g., a cache concept.
  - Then study the performance of different algorithms or algorithm variants as assumptions about these parameters are varied (i.e., are refined into different subconcepts).
- In this regard it appears MMIX can serve as an extremely useful guidepost toward the important hardware concepts to include and vary.

21 Organizing details and summarizing statistics

- Main drawback to introducing and varying hardware parameters: the amount of detail that must be reported to give a fully accurate picture of an algorithm's performance.
- But organization of information using concept webs could help in
  - suppression of details at one level while revealing them fully at deeper levels;
  - summarization, aggregation of statistics.

22 Conclusion

- Algorithm concept hierarchies offer an appealing way to classify and present knowledge of algorithms.
- The set of requirements defining an algorithm concept can include performance requirements.
- At a suitable level of concept refinement these performance requirements can be stated strictly enough that they serve as a useful standard:
  - software library implementors have freedom to tailor implementations to a particular platform
  - application programmers can rely on being able to port their programs from one platform to another without dramatic changes in performance

A Sample Algorithm Concept Descriptions

A.1 Computational Method

A (generic) *computational method* is a method for solving a specific type of problem by means of a finite set of steps operating on *inputs*, which are quantities given to it before execution of the steps begins or during executing, and producing one or more *outputs*, which have a specified relation to the inputs. The number of steps in the method is required to be not only finite but also independent of the inputs. (The program does not grow or shrink in response to the inputs, but it might have different variations for different types of inputs.) The method is also required to be *resource constrained*, which means there are requirements on all operations of all steps of the

method that constrain the resources (time, space) that can used in executions of the method.

Execution of steps may repeat other steps, so that although the set of steps is finite, executions of them may produce an infinite sequence of steps—finite termination is not a requirement (it *is* a requirement of the algorithm (§A.2) concept). Some nonterminating computational methods are useful, such as computer operating systems or event-driven simulation systems. Even though the execution of such methods does not terminate, we are still generally interested in bounding the number of steps taken in producing some partial output (as in proving response-time guarantees for an operating system).

In order to bound the resources—time and space—consumed during an execution of the method, we first need bounds on the resources consumed by indi-

vidual steps. This motivates the resource-constraint requirement on computational methods.

Effectiveness of a computational method is the property that all operations of all steps of the method "must be sufficiently basic that they can in principle be done exactly and in a finite length of time." (Knuth [TAOCP, Vol. 1] adds "by someone using pencil and paper," but it is a philosophical question whether humans have any computational capability beyond the effectiveness of machines.) As defined here, effectiveness of computational methods follows from their resource-constraint requirement.

Definiteness of a computational method is the property that each step of the method "must be precisely defined; the actions to be carried out must be rigorously and unambiguously specified for each case" [Knuth, TAOCP, Vol. 1]. This includes the property

that it must be unambiguous which step, if any, follows the current step in any execution of the method.

Again, resource-constraint requirements place some limitations on just how "indefinite" the steps of a method may be.

A.2 Algorithm

Finiteness of a computational method is the property that the number of steps in any execution of the method must be finite. The finiteness property is also called *termination*, and the method is said to be *terminating*. *Algorithm* is a synonym for *finite computational method*, a computational method (§A.1) with the additional property of finiteness. Every abstraction that belongs to an algorithm concept must have the termination property.

Note that among the abstractions belonging to a computational method concept, some might be terminating while others are nonterminating.

A.3 Algorithm Specialized by Input

This concept is a narrowing of the algorithm (§A.2) concept by restrictions on the form of input. Sub-concepts include set algorithms, sequence algorithms (§A.5), tree algorithms, graph algorithms, algebraic algorithms, etc.

A.4 Algorithm Specialized by Strategy

This concept is a narrowing of the algorithm (§A.2) concept in terms of strategies used in structuring the steps of the algorithm. Subconcepts include Divide-and-Conquer Algorithm (§A.12), Dynamic Programming Algorithm, Greedy Algorithm, etc.

A.5 Sequence Algorithm

A *sequence algorithm* is an algorithm (§A.2) that takes one or more linear sequences as inputs. Thus it is a refinement of Algorithm Specialized by Input (§A.3).

A.6 Comparison Based Sequence Algorithm

A *comparison based sequence algorithm* is an sequence algorithm (§A.5) whose computation depends on comparisons between pair of values in the sequence.

A.7 Index Based Sequence Algorithm

An *index based sequence algorithm* is a sequence algorithm (§A.5) that operates only on the positions within the sequence, independently of the values stored.

A.8 Predicate Based Sequence Algorithm

A *predicate based sequence algorithm* is a sequence algorithm (§A.5) whose computation depends on the results of applying a given predicate to values in the sequence.

A.9 Transform Based Sequence Algorithm

A *transform based sequence algorithm* is a sequence algorithm (§A.5) that applies a given transformation function to values in the sequence.

A.10 Sequence Permuting Algorithm

A *sequence permuting algorithm* is a sequence algorithm (§A.5) whose output is a permutation of the its input.

A.11 Sequence Sorting Algorithm

Input: A sequence of elements in a range [first, last).

Output: A modified sequence of elements in the same range.

Refinement of: Comparison Based (§A.6), Permuting (§A.10), Sequence Algorithm (§A.5).

Effects:

- After execution, the elements in [first, last) are a permutation (§A.10) of the input.
- After execution, the elements in [first, last) are in nondecreasing order according to the comparison operator (§A.6).

A.12 Divide-and-Conquer Algorithm

A *divide-and-conquer algorithm* is an algorithm (§A.2) whose steps are structured according to the following strategy:

1. Construct the output directly and return it, if the input is simple enough. Otherwise:
2. Divide the input into two or more (a finite number) of smaller inputs.
3. Recursively apply the algorithm to each of the smaller inputs produced in the first step.
4. Combine the outputs from the recursive applications to produce the output corresponding to the original input.

This concept is one of many known ways of narrowing the algorithm concept in terms of a strat-

egy (§A.4), which gives a specific structure to the steps of the algorithm.

A.13 Divide-and-Conquer Sequence Algorithm

A *divide-and-conquer sequence algorithm* is a divide-and-conquer algorithm (§A.12) that is also a sequence algorithm (§A.5).

Subconcepts are commonly based on different ways of dividing a sequence up into smaller ones.

A.14 Divide-and-Conquer Sorting Algorithm

A *divide-and-conquer sorting algorithm* is a sorting algorithm (§A.11) that is also a divide-and-conquer sequence algorithm (§A.13).

B Sample Operation Count Table

Table 1: Performance of Introsort, Mergesort, and Heapsort on Random Sequences (Sizes and Operations Counts in Multiples of 1,000)

Size	Algorithm	Comparisons	Assignments	Iterator Ops	Integer Ops	Total Ops
1	Introsort	12.1	9.3	52.7	1.1	75.3
	Mergesort	9.7	14.3	59.1	4.1	87.3
	Heapsort	10.3	15.4	137.1	164.5	327.4
4	Introsort	57.3	43.2	247.5	4.5	352.5
	Mergesort	47.0	65.3	268.2	15.8	396.3
	Heapsort	49.3	69.7	644.4	773.5	1536.9
16	Introsort	261.5	195.3	1122.6	18.5	1597.9
	Mergesort	220.4	293.5	1201.1	62.6	1777.6
	Heapsort	229.2	310.9	2961.2	3557.6	7058.9
64	Introsort	1227.2	870.8	5132.0	73.8	7303.7
	Mergesort	1009.7	1302.2	5317.2	249.9	7879.0
	Heapsort	1044.7	1371.5	13380.1	16087.1	31883.4
256	Introsort	5643.0	3841.5	22988.3	293.7	32766.6
	Mergesort	4551.2	5721.1	23313.4	997.2	34583.0
	Heapsort	4691.2	5998.6	59668.5	71778.8	142137.1
1024	Introsort	24777.7	16827.7	100747.5	1176.1	143529.0
	Mergesort	20250.5	24927.3	101439.7	3992.4	150610.0
	Heapsort	20812.7	26042.3	263250.6	316801.4	626907.1

C

Random Data