Algorithm-Oriented Generic Libraries

David R. Musser* Alexander A. Stepanov†

September 1993

Abstract

We outline an approach to construction of software libraries in which generic algorithms (algorithmic abstractions) play a more central role than in conventional software library technology or in the object-oriented programming paradigm. Our approach is to consider algorithms first, decide what types and access operations they need for efficient execution, and regard the types and operations as formal parameters that can be instantiated in many different ways, as long as the actual parameters satisfy the assumptions on which the correctness and efficiency of the algorithms are based. The means by which instantiation is carried out is language dependent; in the C++ examples in this paper, we instantiate generic algorithms by constructing classes that define the needed types and access operations. By use of such compile time techniques and careful attention to algorithmic issues, it is possible to construct software components of broad utility with no sacrifice of efficiency.

*Rensselaer Polytechnic Institute, Computer Science Department, Troy, New York 12180

†Hewlett-Packard Laboratories, Parallel Algorithms Program, 1501 Page Mill Road, Palo Alto, California 94303

Contents

1 Introduction

The last few years have seen the development of software libraries in which the library components are parameterized by data types and functions, making them more general, or "generic," than components in older libraries. Parameterization is done using compile time mechanisms such as generics or templates (e.g., Booch's Ada and C++ libraries [1, 2]) or preprocessing mechanisms (e.g., Lea's GNU C++ library [6]), achieving greater run-time efficiency than was possible with older methods (for example, passing at run-time the size of data elements and a comparison function to C library routines such as `qsort` or `bsearch`). But in most cases parameters are still restricted to scalar parameters, data types, or functions, and do not include what might be called "container representations"—ways of representing data containers such as sequences, sets, trees, graphs, matrices, etc. (e.g., for operations on sequences, one might have container representations using arrays, linked-lists, ranked red-black trees, etc.). Consequently such libraries may have to reimplement the same algorithm many times, once for each of the possible container representations.

In our approach to software library construction, we allow algorithms to be parameterized not only by scalar values, data types, and functions, but also by container representations. Of course, many algorithms are efficient only with a particular kind of container representation, say linked-lists, but even within this single kind of representation there is a wide variety of concrete ways of setting up node structure, managing storage allocation, and handling error conditions. Many commonly useful operations on sequences, such as inserting, deleting, substituting, concatenating, merging, and searching, can be performed with algorithms that depend for their correct and efficient operation only on a few basic access operations. By expressing the algorithms in terms of these basic access operations and making the access operations parameters, we permit a single expression of the algorithms to be used with any concrete representation of the container.

1.1 Outline of the algorithm-oriented approach

The key steps of our approach to generic library construction are:

- Start with the most efficient known algorithms and data structures, identify container access operations (such as data moves, exchanges, or comparisons) on which the algorithms depend, and abstract (generalize) those operations by determining the minimal behavior they must exhibit in order for the algorithm to perform a useful operation.

- Separately develop various ways of implementing the container access operations using different container representations with different efficiency characteristics, such as using random access or linked structures, with further classification according to use of different processor or storage allocation strategies and different ways of handling errors.

- Practice software reuse within the library itself, by identifying small algorithmic building blocks and implementing them as separate functions from which larger algorithms can be composed.

- Thoroughly document the algorithms in overviews that compare various algorithms and identify favorable contexts for their use and in individual component "data sheets" that describe key attributes that programmers need to know for intelligent selection and proper use.

The intended use of such a library involves several steps of selection and instantiation of components from the library:

- selection of the algorithms to be used;

- selection of a container representation suited to the selected algorithms and other constraints;

- combining the container representation with the algorithms, which essentially consists of instantiating the container access parameters used in the algorithms with types that provide those operations.

- instantiating other parameters of the generic algorithms, such as data types and problem size.

Altogether, this flexibility means a single generic algorithm can have broad utility. Yet efficiency is not sacrificed, because algorithmic efficiency is respected in the design of the algorithms and their recommended uses, and because the container representations and algorithms can be combined without the overhead of subprogram calls (by using inline declarations and templates and/or macros).

1.2 The algorithm-oriented approach in C++

We demonstrated an earlier version of the algorithm-oriented approach in Ada in [7, 8, 9]. In this paper, we illustrate the approach with a number of generic algorithms implemented in C++. These algorithms are part of a library of operations on sequences of values, using array, linked-list, or hybrid array/linked representations of sequences, including partitioning, merging, and sorting operations.

The algorithms in the library are designed to work with a variety of different choices for type of data elements and container representations. The specialization to particular choices occurs at compile time, according to definitions given in the source code of the application program. In C++, we express generic algorithms by means of template function definitions and container representations by template (or ordinary) class definitions. For example, the generic quicksort algorithm given in this paper can be combined with a vector representation of sequences (i.e., using a block of consecutive storage locations) or in a variety of other representations.

Some algorithms have even greater flexibility; consider, for example, the count algorithm, which is defined on a sequence of values and returns the number of elements in the sequence that satisfy a given predicate. It could be used to count the number of elements in a sequence of integers that are positive, for example, by using a predicate on integers which tests whether its argument is positive. The algorithm used by count is simply to consider each element in turn, and for this purpose it uses a `++` operator; this operator can be supplied as either one of the standard ones that advance a pointer, or a user defined one that chases a link in a linked representation. The sequence can thus be represented in a linked-list structure as well as in a block of consecutive locations.

1.3 Type requirements

In describing a generic algorithm, we list certain types as parameters and describe, under the heading of *type requirements*, the interfaces and semantics of operations that those types are required to have. In C++, we can syntactically indicate which types are parameters using template class or template function declarations; however there is no way in the language to express the requirement that the types supply certain operations (other than just by using the operations in function bodies, which doesn't always make clear which types must supply them). Instead, we list operation prototypes and corresponding informal semantic descriptions as part of the documentation of the algorithms. To avoid repetition of these descriptions for each component, we use operation names consistently and collect together the full descriptions of the corresponding semantic and computing time requirements, then abbreviate these in the component data sheets.

In general, the statement of type requirements for generic algorithms is a complex and important subject. For present purposes, we restrict our attention to two types needed for sequence algorithms, namely sequence iterator types and functional types. The requirements on these types are spelled out in the next two subsections.

1.3.1 Iterator types

Iterator types generalize the notion of pointer, encapsulating information about locations in objects. The values held by iterator objects are locations, and iterator operations provide for sequencing through a series of locations and obtaining information from those locations. The choices we make of operator names are consistent with C++ notation for C++ pointer types. Generally, iterators are defined in C++ using the class mechanism, but they may also be pointer types (`T*` where `T` is any built-in or user-defined type).

To give the semantic requirements on sequence iterators, we use the following conventions. We assume that for any (finite) sequence $x_0, x_1, \ldots, x_{n-1}$ there is a sequence of distinct locations $i_0, i_1, \ldots, i_n$, where i_j is the location of x_j for $j = 0, \ldots, n-1$, and i_n is an additional location considered to be "off the end" of the sequence. Each

location i_{j+1} is called the successor of i_j, and i_j is called the predecessor of i_{j+1}, for $j = 0, \ldots, n-1$. An array representation of sequences commonly uses a set of integers in an arithmetic progression as locations (with the off-the-end value just being the next higher integer in the progression following the location of the last element), while a linked-list representation uses a set of pointers (usually with the null pointer as the off-the-end value).

We say that location i_j is *to the left of* location i_k if $j < k$ and is *to the right of* location i_k if $k < j$. Thus it makes sense to speak of the *leftmost* or *rightmost* location with a particular property. We also speak of left-to-right traversal $(i_0, i_1, \ldots)$ or a right-to-left traversal $(i_n, i_{n-1}, \ldots)$.[1]

An operation we require of all iterator types is dereferencing (`operator*()`). In the case of sequence iterators, this operation returns the value x_j when location i_j is currently held by the iterator, $j = 0, \ldots, n-1$. If the iterator holds i_n, the result of dereferencing is undefined. Dereferencing is required to be a constant time operation. In component data sheets, we abbreviate these requirements as

`T& operator*() const`	dereference

where `T` is a type introduced elsewhere in the description. i.e, this entry under **Type requirements** should be taken to mean the requirements, on both functionality and computing time, stated in this paragraph.

We also require that all iterator types provide

`int operator==(Iterator)`	equality check
`int operator!=(Iterator)`	inequality check

where, if `x` holds i_j and `y` holds i_k, `x == y` returns 1 if $j = k$ and 0 otherwise; `x != y` has the opposite meaning. In some cases we further require

`int operator<=(Iterator)`	less than or equal check

where the ordering observed by this operator is `x <= y` returns 1 when $0 \leq j \leq k \leq n$, 0 otherwise.

Sequence iterators must also provide a traversal operation (`operator++()`) for advancing from a location to its successor; in some cases, but not all, we also require an operation (`operator--()`) that decrements from a location to its predecessor. Since C++ allows different definitions to be given for either prefix or postfix applications of these operators [4], we define the requirements on both:

1. For $j = 0, \ldots, n-1$, if the iterator `x` holds location i_j then `x++` or `++x` causes it to hold i_{j+1}; the value returned by `x++` is i_j while that returned by `++x` is i_{j+1}. If `x` holds i_n, then the effect and return values of both `x++` or `++x` are undefined.

[1] In this paper we will also use the terms "upward" synonymously with left-to-right and "downward" with right-to-left. Note that none of these definitions make use of an ordering on the locations values themselves; thus even though the addresses used in a linked list representation could be compared as integers, we do not rely on that kind of comparison.

2. For $j = 1, \ldots, n$, if the iterator `x` holds location i_j then `x--` or `--x` causes it to hold i_{j-1}; the value returned by `x--` is i_j while that returned by `--x` is i_{j-1}. If `x` holds i_0, then the effect and return values of both `x--` or `--x` are undefined.

3. All four of these operations must be constant time operations.

These requirements are abbreviated in **Type requirements** with the following kinds of entries

`Iterator operator++(int)`	postfix increment
`Iterator operator++()`	prefix increment
`Iterator operator--(int)`	postfix decrement
`Iterator operator--()`	prefix decrement

In some cases we may also require a sequence iterator to provide

`Iterator operator+(int)`	addition of an int
`Iterator operator-(int)`	subtraction of an int

with the meaning that for $j = 0, \ldots, n$, if `x` holds location i_j, then

1. if $0 \leq k \leq n - j$ then the location returned by $x + k$ is i_{j+k}, and

2. if $0 \leq k \leq j$ then the location returned by $x - k$ is i_{j-k}.

The maximum time used by these operations must be no more than linear in k; for some algorithms, we might require constant time (or we might give separate computing time analyses under both assumptions). Similarly, we might require

`Iterator operator+=(int)`	increment by an int
`Iterator operator-=(int)`	decrement by an int

Finally, some algorithms require a sequence iterator to provide

`int operator-(Iterator)`	subtraction

with the meaning that for $0 \leq j, k \leq n$ if `x` holds location i_j and `y` holds location i_k, then the integer returned by `x - y` is $j - k$.[2] The maximum time used in computing `x - y` must be no more than linear in $j - k$; for some algorithms we might require constant time (or we might give separate computing time analyses under both assumptions).

All of these requirements are fulfilled by pointer types in C++ (or C). They are also fulfilled by a singly-linked list class in our library, except that it omits `--`, `-`, and `<=` since these operations cannot be done in constant time on singly-linked lists. Our library also contains a doubly-linked list class that does provide `--` and `-`, so that algorithms that need efficient backward traversal can be combined with this iterator.[3] Another example of an iterator type is given in Section 3.

[2]Note that this is defined even if `x` or `y` holds i_n, the "off-the-end" value; for example, if `x` hold i_n and `y` holds i_0 then `x - y` returns n, the length of the sequence.

[3]The requirements on `--` and `-` for a doubly linked representation imply that it is necessary to use a non-null value as the off-the-end location i_n to enable backward traversal to work even when starting from i_n.

1.3.2 Functional types

Some of the algorithms to be described have function parameters, such as predicates. Rather than following the common C/C++ programming practice of passing a pointer to a function, we can produce more efficient code by taking advantage of the ability in C++ to overload the function call operator, `operator()`.

For example, our sorting algorithms are parameterized by a type that is required to provide a function to compare two values x and y of some type T and return either a negative integer, 0, or a positive integer according to whether x is less than, equal to, or greater than y (in some total ordering of T). The function must execute in constant time. In the type requirements on data sheets, these requirements are abbreviated by an entry for a type parameter called a *comparator*, of the form

`int operator()(T, T)`	compare two `T` values, 3-way

Such a comparator type can be defined in C++ by a class definition such as

```
class intComparator {
public:
  intComparator(){}
  int operator()(int x, int y) {return x - y;}
};
```

which in this case defines `operator()` in terms of subtraction. For a different way of defining comparison, only the body of the `operator()` definition would be changed. More generally, one could use a template class definition such as

```
template <class T>
class Comparator {
public:
  Comparator(){}
  int operator()(T x, T y) {return x - y;}
};
```

which provides a definition that can be used with any of the C++ signed integer types. A comparator type instance for `T = int` can then be constructed using the constructor member of the class definition, as in the following calls of the `quickSort` function described in Section 2.6

```
quickSort(first, last, intComparator());
```

or

```
quickSort(first, last, Comparator<int>());
```

In general, we say that a type is a *functional type* if it is defined by a C++ class that provides one or more definitions of the function call operator, but no data members. Note that instances of a functional type do not require any storage. Since

a C++ compiler can inline the definition of the function at the site of calls, using functional types not only avoids the overhead of an indirect function call (as occurs when a pointer to a function is passed), it even eliminates the cost of a direct call.

1.4 Complexity requirements

It is important to recognize that we place certain requirements on the complexities of access operations.

For sequence iterator operations, we require that the cost of `++` and `--` is constant; that is, going to the next or previous position is not costly. That is why we do not provide `--` for singly linked lists. However, we do not place such a requirement on `+` and `-`; they are assumed to be linear in the worst case. Indeed, in our opinion, the main difference between linked-lists and vectors—forgetting for a moment mutative operations such as insert—is exactly that vectors have a constant time operation `+` on their iterator type while linked-lists have a linear time one. In other words, vectors allow for cheap long jumps, while linked-lists favor one-by-one accesses.

For the type of data values stored, we also require constant time operations for assignment and comparison. But within these requirements, there is still considerable room for differences in the complexity of different operations—in one case we might have assignment costing several times what a comparison costs, in other cases the opposite, and in still other cases they might cost about the same. These differing situations sometimes need to be considered in choosing algorithms; consequently, in the discussions of complexity in the overviews and data sheets in the following sections, we try to include these considerations.

2 Examples of generic sequence algorithms

To illustrate the algorithm-oriented approach in C++, we give a small sample of algorithms for operations on sequences, specifically partitioning and sorting algorithms. Some of these algorithms require iterator types that provide the full set of iterator operations described in Section 1.3.1. The iterator operations must be constant time operations, except perhaps long jumps (`operator+(int)` and `operator-(int)`) which are permitted to take linear time since the use of long jumps is limited.

As noted in Section 1.3.1, these type requirements are naturally satisfied by standard C++ pointer types. When such types are used as iterators with the algorithms, the resulting versions are as efficient as any non-generic, hand-tailored version could be for those types. An example is given in Section 3 of a user-defined iterator type with which these generic algorithms can also be combined to produce efficient algorithms.

The generic algorithms presented in this section also serve to illustrate the coding and documentation conventions we have chosen to use. We begin with an overview and comparison of the algorithms. Two sorting algorithms, `insertionSort` and

`quickSort`, are included. Both operate in-place (the result is placed in the storage occupied by the original sequence and only a constant or logarithmic amount of extra storage is required). The first has order n^2 worst case computing time on a vector of length n, but runs in linear time and is the sorting algorithm of choice in special circumstances, as detailed on its data sheet. It is a stable sort, in the sense that elements that compare equal appear in the result in the same relative order as in the original sequence. The second is based on Hoare's quicksort algorithm and has expected time of $O(n \log n)$; taking $O(n^2)$ time is possible but occurs only with extremely low probability. This algorithm makes more comparisons but makes substantially fewer data moves than merge sort and thus is recommended in settings where stable sorting is not required and the cost of a comparison is not substantially more than that of a data move. Another sorting related generic algorithm documented here is `unguardedPartition`, which permutes a sequence into two subsequences, one containing elements that compare less than or equal a given value, and the other containing elements that compare greater than or equal the value. It is used in implementing `quickSort` (as is `insertionSort`) and other components.[4]

Concrete versions of these algorithms may be found in standard references, e.g., [3, 5], and the research literature, e.g., [11]. In constructing generic algorithms, one can often benefit from this prior work, but one must be careful to ensure that optimizations can still be done in a general setting and, if so, that they remain optimizations in most, if not all, settings. For example, use of some special sentinel value in an extra array position to stop a search, as is typically done in coding insertion sort in order to have the fastest possible inner loop, must be modified since in some instances an extra array position might not be available. We could have just abandoned the sentinel technique and provided an algorithm that is general but whose instances are in some cases less efficient than hand-tailored code. Instead, we provide different versions of crucial routines, in which we use the sentinel technique in one and not in the other and limit the use of the non-sentinel, less efficient version to a case with a small number of elements.

We now give the data sheets for the algorithms discussed above and other generic algorithms used in their implementation.

2.1 moveBackward

Declaration

```
template <class Iterator1, class Iterator2>
inline void moveBackward(Iterator1 first, Iterator1 last,
    Iterator2 destination);
```

[4]The use of partition and insertion sort algorithms as subprograms, rather than inline incorporation of special versions, illustrates our preference for modular construction of the library's components. One of our goals is to demonstrate that such modularization can be done without sacrificing efficiency.

Description Copies the values in the sequence in locations `first`, ..., `last` $-$ 1 to locations `destination` $- n$, ..., `destination` $-$ 1, where n = `last` $-$ `first`. The source and destination ranges may overlap if `destination` $\geq$ `last`.

See Also `move`, `reverseCopy`, `swapRange`

Time Complexity Linear. The number of value assignments performed is n = `last` $-$ `first`.

Space Complexity Constant

Mutative? Yes

Type Requirements For some types `T1` and `T2` (note that these are not parameters) such that `T2` provides

`T2& operator=(T1)`	assignment

type `Iterator1` must provide

`T1& operator*() const`	dereference
`Iterator1 operator--()`	prefix decrement
`int operator!=(Iterator1)`	inequality check

and type `Iterator2` must provide

`T2& operator*() const`	dereference
`Iterator2 operator--()`	prefix decrement

Details

1. Note that `destination` refers to the upper boundary of the range to which the values are copied (whereas with `move` it refers to the lower boundary).

2. Among the uses of this function is shifting a sequence one or more locations to the right ("backwards" refers to the order in which elements are moved, not to the direction in which the whole sequence is shifted), when `Iterator2` is the same type as `Iterator1` and `destination` occurs to the right of `last`.

3. Since types `Iterator1` and `Iterator2` need not be the same, this function can be used for copying information from one sequence representation (such as array) to another (such as linked-list).

Implementation

```
template <class Iterator1, class Iterator2>
inline void moveBackward(Iterator1 first, Iterator1 last,
    Iterator2 destination)
{
    while (first != last)
        *--destination = *--last;
}
```

Implementation Notes The values in locations `last` $-$ 1 down to and including `first` are copied sequentially to locations `destination` $-$ 1 down to and including `destination` $- n$.

2.2 linearInsert

Declaration

```
template <class Iterator, class T, class Comparator>
inline Iterator unguardedLinearInsert(Iterator last, T value,
    Comparator compare);

template <class Iterator, class T, class Comparator>
Iterator linearInsert(Iterator first, Iterator last, T value,
    Comparator compare);
```

Description Either function inserts `value` in an ascendingly sorted sequence so that the result is still ascendingly sorted (according to `compare`).

See Also `binaryInsert` `insertionSort`

Time Complexity Linear. If k is the number of elements in locations to the left of `last` that are greater than `value`, the number of data assignments and the number of comparisons are each $k + 1$.

Space Complexity Constant

Mutative? Yes

Type Requirements Type `T` must provide

`T& operator=(T)`	assignment

type `Iterator` must provide

`T& operator*() const`	dereference
`Iterator operator--()`	prefix decrement
`Iterator operator++()`	prefix increment
`int operator!=(Iterator)`	inequality check

and type `Comparator` must provide

`int operator()(T, T)`	compare two `T` values, 3-way

Details

1. It must be possible to assign to location `last`, as it is used to hold a value of the resulting sequence.

2. `unguardedLinearInsert` assumes there is some location to the left of `last` that holds a value no larger than `value`; if the rightmost such value is in location p, the sequence affected is that in locations $p, p+1, \ldots$ `last`, and it inserts `value` in location p after shifting the values in the range p through `last` -1 to the right by one location . If the sequence in locations $p, p+1, \ldots$ `last` -1 was previously in ascending order according to `compare`, then the resulting sequence in locations $p, \ldots,$ `last` is also in ascending order according to `compare`.

3. `linearInsert` assumes that `first` $\neq$ `last`, and inserts `value` in one of the locations `first`, ..., `last` -1, after right-shifting by one all the values from the insertion point to the end. If the values in locations `first`, ..., `last` -1 were previously in ascending order according to `compare`, `value` is inserted in the proper place to make all the values in locations `first`, ..., `last` in ascending order according to `compare`.

Implementation

```
template <class Iterator, class T, class Comparator>
inline Iterator unguardedLinearInsert(Iterator last, T value,
    Comparator compare)
{
    Iterator previous = last;
    while (compare(value, *--previous) < 0) {
        *last = *previous;
        last = previous;
    }
    *last = value;
    return last;
}
```

```
template <class Iterator, class T, class Comparator>
Iterator linearInsert(Iterator first, Iterator last, T value,
    Comparator compare)
{
    if (compare(value, *first) >= 0)
        return unguardedLinearInsert(last, value, compare);
    Iterator next = last;
    moveBackward(first, last, ++next);
    *first = value;
    return first;
}
```

Implementation Notes `unguardedLinearInsert` scans to the left looking for a value no larger than `value` and shifting values one location to the right as the scan proceeds. It depends on there being a location before `last` that contains a value no larger than `value`; otherwise, it would loop forever.

In `linearInsert`, if `value` compares nonnegative with the value in location `first`, the insertion is done using `unguardedLinearInsert` (since the latter value serves to stop the iteration); otherwise, `value` is placed in the `first` location after shifting upward all the values in the sequence.

2.3 insertionSort, unguardedInsertionSort, thresholdInsertionSort

Declaration

```
template <class Iterator, class Comparator>
void unguardedInsertionSort(Iterator first, Iterator last,
    Comparator compare);

template <class Iterator, class Comparator>
void insertionSort(Iterator first, Iterator last,
    Comparator compare);

template <class Iterator, class Comparator>
void thresholdInsertionSort(Iterator first, Iterator last,
    int threshold, Comparator compare);
```

Description `insertionSort` sorts the sequence in locations `first`, ..., `last` $-$ 1 in place, into ascending order according to the ordering determined by `compare`. `unguardedInsertionSort` is faster but possibly includes additional locations preceding `first` in the sequence sorted (see details). `thresholdInsertionSort` is faster than `insertionSort` but assumes the minimum value in locations `first`, ..., `last`-1

occurs in the `threshold` lowest locations. These functions are not recommended for general use but are a good choice for sorting short or "almost sorted" sequences.

See Also `quickSort`, `mergeSort`

Time Complexity Quadratic, in the average and worst cases. On the average the number of `compare` operations performed is $n^2/4$ and the number of `T` assignment operations is the same, where $n = \texttt{last} - \texttt{first}$. For most inputs both are very slow compared to the best sorting algorithms. However, these functions are quite fast for small sequences ($n \leq 16$ or so) or for large ones that are "almost sorted" in one of the following senses: (1) the number of elements out of order is small, or (2) the average distance between the original location of an element and its final destination is small. (By small we mean less than about 16.) For such sequences the worst case time is linear in the size of the sequence.

Space Complexity Constant

Mutative? Yes

Type Requirements Type `T` must provide

`T& operator=(T)`	assignment

type `Iterator` must provide

`T& operator*() const`	dereference
`Iterator operator--()`	prefix decrement
`Iterator operator++()`	prefix increment
`Iterator operator++(int)`	postfix increment
`Iterator operator+(int)`	addition of an int
`int operator-(Iterator)`	subtraction
`int operator==(Iterator)`	equality check
`int operator!=(Iterator)`	inequality check

and type `Comparator` must provide

`int operator()(T, T)`	compare two `T` values, 3-way

Details

1. All three versions are stable sorts; that is, the relative order of elements that are equal (according to `compare`) is preserved.

2. `unguardedInsertionSort` is the fastest version (has the smallest coefficient in its computing time bound), but it correctly sorts only under an extra assumption: that for some location $p \leq$ `first` the sequence in locations $p, p+1, \ldots,$ `first` -1 is already sorted and the value in location p is a minimum for the *extended* sequence in locations $p, p+1, \ldots,$ `last` -1. The result is that this extended sequence is sorted into ascending order. Note if $p \neq$ `first`, the sequence `unguardedInsertionSort` leaves in locations `first`, $\ldots,$ `last` -1 is in ascending order but is not a permutation of the values originally in those locations (some values change places with those in locations $p, p+1, \ldots,$ `first` -1).

Implementation

```
template <class Iterator, class Comparator>
void insertionSort(Iterator first, Iterator last,
    Comparator compare)
{
    if ((first == last) || (first + 1 == last)) return;
    for (Iterator i = first + 1; i != last; i++)
        (void)linearInsert(first, i, *i, compare);
}
template <class Iterator, class Comparator>
void unguardedInsertionSort(Iterator first, Iterator last,
    Comparator compare)
{
    if (first == last) return;
    for (Iterator i = first; i != last; i++)
        (void)unguardedLinearInsert(i, *i, compare);
}
template <class Iterator, class Comparator>
void thresholdInsertionSort(Iterator first, Iterator last,
    int threshold, Comparator compare)
{
    if (last - first > threshold) {
        insertionSort(first, first + threshold, compare);
        unguardedInsertionSort(first + threshold, last, compare);
    } else
        insertionSort(first, last, compare);
}
```

Implementation Notes The basic idea is to scan the sequence from left to right and insert the current element into its proper place among the previously scanned and already sorted elements. Each insertion just involves a scan from the current

location to the left, shifting elements right by one location as the scan proceeds, so that there will be a place for the element being inserted.

For greater speed `unguardedInsertionSort` uses `unguardedLinearInsert`, which omits any check for the scan passing the left end, `first`. Hence it depends on the assumptions stated in Detail 2 being satisfied.

Advantage of `unguardedInsertionSort` is taken by `thresholdInsertionSort`, which uses `insertionSort` to sort the first `threshold` values. Unguarded scans may then be used for the rest of the sequence, since by the assumption stated in the Description and the results of `insertionSort`, the assumptions described in Detail 2 are satisfied for the call to `unguardedInsertionSort`.

2.4 medianOf3Select

Declaration

```
template <class Iterator, class Comparator>
inline Iterator medianOf3Select(Iterator first, Iterator last,
    Comparator compare);
```

Description Returns the location of the second largest of three values in the sequence from `first` to `last` (the values in the leftmost, rightmost, and middle locations), using the ordering determined by `compare`.

Time Complexity Constant.

Space Complexity Constant.

Mutative? No

Type Requirements For some type `T` (note that `T` is not a parameter) type `Iterator` must provide

`T& operator*() const`	dereference
`Iterator operator+(int)`	addition of an int
`Iterator operator--(int)`	postfix decrement
`ptrdiff_t operator-(Iterator)`	subtraction

and type `Comparator` must provide

`int operator()(T, T)`	compare two `T` values, 3-way

Details The three values examined are those in locations `first`, `first` + (`last` − `first`)/2, and `last` − 1.

Implementation

```
template <class Iterator, class Comparator>
inline Iterator medianOf3Select(Iterator first, Iterator last,
    Comparator compare)
{
    Iterator middle = first + ((last - first) >> 1);
    last--;
    if (compare(*first, *middle) <= 0)
        if (compare(*middle, *last) <= 0)
            return middle;
        else if (compare(*first, *last) <= 0)
            return last;
        else
            return first;
    else if (compare(*first, *last) <= 0)
        return first;
    else if (compare(*middle, *last) <= 0)
        return last;
    else
        return middle;
}
```

2.5 unguardedPartition

Declaration

```
template <class Iterator, class T, class Comparator>
inline Iterator unguardedPartition(Iterator first, Iterator last,
    T pivot, Comparator compare);
```

Description Permutes the sequence in locations `first`, ..., `last`-1 in place, partitioning it into two subsequences such that `compare`$(*i,$`pivot`$) \leq 0$ for all locations i in the left subsequence and `compare`$(*j,$`pivot`$) \geq 0$ for all locations j in the right subsequence (if any). Returns the location that marks the beginning of the right subsequence.

See Also `quickSort`, `select`, `partition`, `stablePartition`

Time Complexity Linear. The number of comparisons performed (using `compare`) is either $n+1$ or $n+2$, where $n =$ `last` $-$ `first`, and the number of `swap` operations is at most $\lfloor n/2 \rfloor$.

Space Complexity Constant

Mutative? Yes

Type Requirements Type `T` must provide

`T& operator=(T)`	assignment

type `Iterator` must provide

`T& operator*() const`	dereference
`Iterator operator++(int)`	postfix increment
`Iterator operator--(int)`	postfix decrement
`int operator<=(Iterator)`	less than or equal check

and type `Comparator` must provide

`int operator()(T, T)`	compare two `T` values, 3-way

Details

1. There must at least one location i for which `compare`$(*i,$`pivot`$) \leq 0$ and at least one location j for which `compare`$(*j,$`pivot`$) \geq 0$. These conditions are met if there is at least one location i in `first`, ..., `last`-1 for which

$$\texttt{compare}(*i, \texttt{pivot}) = 0.$$

2. If j is the location returned, then `first` $\leq j \leq$ `last`. (Thus, either subsequence may be empty.)

3. Unlike some versions of partitioning, it is not guaranteed that

$$\texttt{compare}(*j, \texttt{pivot}) = 0,$$

   where j is the location returned.

4. The permutation is not stable. (Stability in this case would mean that within each subsequence the relative order of the elements is the same as in the original sequence.) If stability is necessary, see `stablePartition`.

Implementation

```
template <class T>
inline void swap(T& a, T& b)
{
    T tmp = a;
    a = b;
    b = tmp;
```

```
}
template <class Iterator, class T, class Comparator>
inline Iterator unguardedPartition(Iterator first, Iterator last,
         T pivot, Comparator compare)
{
    while (1) {
        while (compare(*first, pivot) < 0) first++;
        last--;
        while (compare(*last, pivot) > 0) last--;
        if (last <= first) return first;
        swap(*first, *last);
        first++;
    }
}
```

Implementation Notes The basic idea of the algorithm is to search from the left for an element that compares non-negative with `pivot`, search from the right for an element that compares non-positive with `pivot`, and, provided the iterators haven't converged or crossed, swap the elements found; then the iterators are moved one step further and the process is repeated. The following points are important for the correctness and utility of the algorithm:

1. The inner loops need no check for running off the end of the sequence: by the assumption described in Detail 1, for each loop there is some element that will stop it, and after a swap is performed, there are still elements in locations to stop both loops.

2. As coded, the algorithm sometimes swaps elements that compare equal, which might seem unnecessary. But avoiding this would require adding checks in the loops for the iterators crossing, and, of more concern, it would also mean that for a sequence with all equal elements `quickSort` would obtain partitionings into 1 and $k-1$ elements, for $k = n, n-1, \ldots$, which means that `quickSort` would take order n^2 steps. The code as given results in a split into two equal parts, so that `quickSort` only takes order $n \log n$ time on such inputs.

2.6 quickSort

Declaration

```
template <class Iterator, class Comparator>
void quickSort(Iterator first, Iterator last, Comparator compare);
```

Description Sorts the sequence in place, into ascending order according to the ordering determined by `compare`. For most inputs, this is one of the fastest sorting algorithms, but for some inputs it is unnacceptably slow. (See Time Complexity).

See Also `mergeSort`, `heapSort`, `insertionSort`

Time Complexity Order $n \log n$, on the average, where $n =$ `last` $-$ `first`. Quadratic in the worst case, but this behavior is highly improbable. Recommended in cases where worst case performance is not critical, stable sorting is not required, and the cost of a comparison (using `compare`) is not too high relative to that of a data move.

Space Complexity Order $\log n$, in the average and worst cases (stack space for recursive calls).

Mutative? Yes

Type Requirements For some type `T` (note that `T` is not a parameter) that provides

`T& operator=(T)`	assignment

type `Iterator` must provide

`T& operator*() const`	dereference
`Iterator operator++()`	prefix increment
`Iterator operator++(int)`	postfix increment
`Iterator operator+(int)`	addition of an int
`Iterator operator--()`	prefix decrement
`Iterator operator--(int)`	postfix decrement
`int operator-(Iterator)`	subtraction of an Iterator
`int operator<=(Iterator)`	less than or equal check

and type `Comparator` must provide

`int operator()(T, T)`	compare two `T` values, 3-way

Details This not a stable sort; that is, the relative order of elements that are equal (according to `compare`) is not preserved. If stability is necessary, see `mergeSort` (which, however, is not an in-place sort).

Implementation

```
#ifndef QUICKSORT_THRESHOLD
#define QUICKSORT_THRESHOLD 16
#endif

template <class Iterator, class Comparator>
static void quickSortLoop(Iterator first, Iterator last,
```

```
    Comparator compare)
{
    while (last - first > QUICKSORT_THRESHOLD) {
        Iterator partition = unguardedPartition(first, last,
            *medianOf3Select(first, last, compare), compare);
        if (partition - first >= last - partition) {
            quickSortLoop(partition, last, compare);
            last = partition;
        } else {
            quickSortLoop(first, partition, compare);
            first = partition;
        }
    }
}
template <class Iterator, class Comparator>
void quickSort(Iterator first, Iterator last, Comparator compare)
{
   quickSortLoop(first, last, compare);
   if (QUICKSORT_THRESHOLD > 1)
       thresholdInsertionSort(first, last,
           QUICKSORT_THRESHOLD, compare);
}
```

Implementation Notes This divide-and-conquer algorithm first partitions the sequence into two parts (working in-place) such that all of the elements in the left part are less than or equal to all of the elements in the right part. It then repeats the partitioning in each of the two parts, continuing in this way until it has achieved a sequence of small partitions in which every element in each partition is less than or equal to all of the elements in the next partition to its right. Then, insertion sort is used to finish putting the elements in order. The algorithm achieves high efficiency because the partitioning step is fast and usually breaks its input into two parts of roughly equal size, and because insertion sort works in linear time on the type of input that quicksort presents to it.

1. The algorithm is expressed using recursion, but the overhead of recursion is kept small by recursing on only one of the two subsequences produced by a partitioning, with the other taken care of iteratively.

2. The recursive calls and iterations both stop when subsequence length drops below a threshold; `thresholdInsertionSort` is used to finish. The identifier `QUICKSORT_THRESHOLD` controls the switch-over; the value 16 is used unless `QUICKSORT_THRESHOLD` is `#defined` as a different value.

3. The final insertion sorting takes only linear time, since no element is more than `QUICKSORT_THRESHOLD` locations out of place. It is correct to use `thresholdInsertionSort` (as opposed to the slower `insertionSort`) since `quicksortLoop` guarantees that the minimum value for the entire sequence occurs in the first `QUICKSORT_THRESHOLD` locations.

4. In the code `if (partition - first >= last - partition)` we choose the smaller of the two subsequences to recurse on: since the smaller must be no more than half the length of the current subsequence, the number of stack frames at any one time due to recursion is no more than $\log_2 n$.

5. There can be up to $n -$ `threshold` partitionings, on sequences of length $n, n - 1, \ldots,$ `threshold` $+ 1$, if each partitioning puts only one element on one side of the partition. This yields the order n^2 worst case time. The median-of-three method of choosing the pivot element makes a long series of such unbalanced partitions extremely unlikely.

6. For partitioning, `unguardedPartition` is used, which exchanges elements even when they are equal according to `compare`. This technique avoids unbalanced partitionings that would otherwise occur when there are many equal elements. Such a sequence is sorted in order $n \log n$ time.

3 An example of a sequence iterator type

The generic algorithms discussed in the previous section can be used not only with the builtin C++ pointer types for the iterators, but with any user-defined type that meets all of the requirements specified in Section 1.3.1. In this section, we present an example of such an iterator type, one that can for example be used to cause a two-dimensional array `T a[m][n]` to appear as a sequence of elements in column order. C++ arrays are stored in row order, and thus just using `T*` as an iterator type allows, for example, using `quickSort` to sort the elements of `a` in row order:[5]

```
const size_t rows = 11;
const size_t cols = 5;
void main() {
   int a[rows][cols];
   int* k(&a[0][0]);
   quickSort(k, k + rows * cols, Comparator<int>());
```

[5]That is, to permute the elements so they are in increasing order when scanned by rows:

$$a[0][0] \le a[0][1] \le \cdots \le a[0][n-1] \le a[1][0] \le a[1][1] \le \cdots.$$

where class `Comparator` is defined as in Section 1.3.2.

To allow our algorithms to work with the sequence of elements in column order, the following class definition provides, for example, a `++` operation that advances from the location of $a[i][j]$ to that of $a[i+1][j]$ if $i < m-1$ or to $a[0][j+1]$ if $i = m-1$. With this iterator type, sorting the array with `quickSort` permutes the array elements to be in increasing order when scanned by columns:

$$a[0][0] \leq a[1][0] \leq \cdots \leq a[m-1][0] \leq a[0][1] \leq a[1][1] \leq \cdots.$$

3.1 Transpose iterator

Declaration

```
template <class Iterator, class T, ptrdiff_t dim1, ptrdiff_t dim2>
class Transpose;
```

Description From a given iterator type, `Iterator`, and integers `dim1` $= m$ and `dim2` $= n$, this class defines a new iterator type that causes the the first mn locations $l_0, l_1, \ldots, l_{mn-1}$ produced by `Iterator` to appear in the *transposed* order

$$\begin{array}{llllll} l_0, & l_n, & l_{2n}, & \ldots, & l_{(m-1)n}, \\ l_1, & l_{n+1}, & l_{2n+1}, & \ldots, & l_{(m-1)n+1}, \\ & \ldots & & & \\ l_{n-1}, & l_{2n-1}, & l_{3n-1} & \ldots & l_{mn-1} \end{array}$$

It can be used for example to cause a two-dimensional array stored in row order to appear as a sequence of elements in column order, or vice versa.

Time Complexity All operations are constant time, provided all `Iterator` operations are constant time. Dereferencing, with `operator*`, is just as fast as for `Iterator`, while the `++`, `--`, `+` and `-` operations are somewhat slower than the corresponding `Iterator` operations.

Space Complexity Constant.

Type Requirements Type `Iterator` must provide

`int operator==(Iterator)`	equality check
`int operator!=(Iterator)`	inequality check
`int operator<=(Iterator)`	less than or equal check
`T& operator*() const`	dereference
`Iterator operator++()`	prefix increment
`Iterator operator++(int)`	postfix increment
`Iterator operator--()`	prefix decrement
`Iterator operator--(int)`	postfix decrement
`Iterator operator+=(int)`	increment by an int
`Iterator operator-=(int)`	decrement by an int
`Iterator operator+(int)`	addition of an int
`Iterator operator-(int)`	subtraction of an int
`int operator-(Iterator)`	subtraction

Provides This class provides the same operations as required of `Iterator`, plus

`Transpose<Iterator, T, dim1, dim2>(Iterator)`	constructor

Details

1. `Iterator` must be capable of producing at least $mn+1$ locations, $l_0, l_1, \ldots, l_{mn-1}, l_{mn}$, where l_{mn} is considered by `Transpose` to be an off-the-end location (whether it actually is for `Iterator` or not). l_{mn} is also used as the off-the-end location for the new iterator.

2. Given an $m \times n$ C++ array `T a[m][n]`, a declaration of the form

   ```
   Transpose<T, m, n> i(&a[0][0])
   ```

   sets up `i` to iterate through the array in column order, i.e., varying the index of the first dimension most rapidly:

   $$a[0][0], a[1][0], \ldots, a[m-1][0], a[0][1], a[1][1], \ldots,$$

3. Conversely, if `b` holds the first location of an $m \times n$ array stored in column order, for example an array created by a Fortran subprogram, a declaration of the form `Transpose<T, n, m> i(b)` sets up `i` to iterate through the array in row order.

Implementation

```
template <class Iterator, class T, ptrdiff_t dim1, ptrdiff_t dim2>
class Transpose {
```

```
    Iterator first, last, current;
protected:
    void advance() {
      if (current - first <= last - first - (dim2 + 1))
         current += dim2;
      else if (current == last - 1)
         current = last;
      else
         current -= last - first - (dim2 + 1);
    }
    void retreat() {
      if (current == last)
         current--;
      else if (dim2 <= current - first)
         current -= dim2;
      else
         current += last - first - (dim2 + 1);
    }
public:
    Transpose(Iterator base) : first(base), last(base+dim1*dim2),
      current(base) {}
    Transpose(const Transpose& t) :
       first(t.first), last(t.last), current(t.current) {}
    T& operator*() const {
      return *current;
    }
    int operator==(const Transpose<Iterator, T, dim1, dim2>& iterator)
         const {
      return current == iterator.current;
    }
    int operator!=(const Transpose<Iterator, T, dim1, dim2>& iterator)
         const {
      return current != iterator.current;
    }
    int operator<=(const Transpose<Iterator, T, dim1, dim2>& iterator)
         const {
      return 0 <= iterator - *this;
    }
    Transpose<Iterator, T, dim1, dim2> operator++() {    // prefix ++
      advance();
      return *this;
    }
```

```
Transpose<Iterator, T, dim1, dim2> operator++(int) { // postfix ++
  Transpose<Iterator, T, dim1, dim2> tmp = *this;
  advance();
  return tmp;
}
Transpose<Iterator, T, dim1, dim2> operator--() {    // prefix --
  retreat();
  return *this;
}
Transpose<Iterator, T, dim1, dim2> operator--(int) { // postfix --
  Transpose<Iterator, T, dim1, dim2> tmp = *this;
  retreat();
  return tmp;
}
Transpose<Iterator, T, dim1, dim2> operator+=(ptrdiff_t k) {
  if (current == last) {
     current = first;
     k += last - first;
  }
  ptrdiff_t i = (current - first) / dim2;
  ptrdiff_t j = (current - first) % dim2;
  ptrdiff_t i1 = (i + k) % dim1;
  ptrdiff_t v = (i + k) / dim1;
  if (i1 < 0) {i1 += dim1; v--;}
  if (i1 >= dim1 || j + v >= dim2)
     current = last;
  else
     current = first + i1 * dim2 + j + v;
  return *this;
}
Transpose<Iterator, T, dim1, dim2> operator-=(ptrdiff_t k) {
  operator+=(-k);
  return *this;
}
Transpose<Iterator, T, dim1, dim2> operator+(ptrdiff_t k) const {
  Transpose<Iterator, T, dim1, dim2> tmp = *this;
  return tmp += k;
}
Transpose<Iterator, T, dim1, dim2> operator-(ptrdiff_t k) const {
  Transpose<Iterator, T, dim1, dim2> tmp = *this;
  return tmp -= k;
}
```

```
    ptrdiff_t operator-(const Transpose<Iterator, T, dim1, dim2>& iterator)
      const;
};
template <class Iterator, class T, int dim1, int dim2>
ptrdiff_t
Transpose<Iterator, T, dim1, dim2>::
 operator-(const Transpose<Iterator, T, dim1, dim2>& iterator)
   const {
      Iterator c1 = current;
      Iterator c = iterator.current;
      if (c1 == last)
         if (c == first)
            return last - first;
         else {c1--; c--;}
      int i1 = (c1 - first) / dim2;
      int j1 = (c1 - first) % dim2;
      int i = (c - first) / dim2;
      int j = (c - first) % dim2;
      return i1 - i + (j1 - j) * dim1;
}
```

Implementation Notes The class maintains an `Iterator` location in `current` and uses it to compute the next new location requested. If `current` holds l_p, where $0 \le p < mn$ and $p = in + j$ with $0 \le j < n$, then its k-successor in the transposed order is the `Iterator` location

$$\texttt{first} + (i + k \bmod m)n + j + \lfloor (i + k)/m \rfloor.$$

This formula is correct even for $k < 0$ (it then gives the $(-k)$-th predecessor). One complicating factor in the code in `operator+=(ptrdiff_t k)` is that if $k < 0$ then the C++ function `%` may give a negative remainder, not the nonnegative remainder always given by the mod function, and the C++ function `/` may correspondingly truncate toward 0 rather than giving the value the floor function would give (C++ implementations are allowed to use whatever the target hardware does). Thus a check for a negative remainder is made and the remainder and quotient are adjusted accordingly.

The code for `advance` and `retreat`, which are used in implementing `++` and `--`, is based on the same formula, but is optimized for faster computation. In both these and the long jump functions, care is taken to treat the special case of l_{mn} properly.

In order to use `quickSort` to sort an array in column order, it is just necessary to set up an iterator of the `Transpose` type and use it in a call of `quickSort`:

```
const size_t rows = 11;
const size_t cols = 5;
void main() {
   int a[rows][cols];
   typedef Transpose<int,(ptrdiff_t)rows,(ptrdiff_t)cols> TransType;
   Transtype k(&a[0][0]);
   quickSort(k, k + rows * cols, Comparator<int>());
```

`Transpose` is an example of an *iterator transformer*, an iterator type that is itself parameterized by an iterator. Such transformers can composed, if the functions provided by one iterator meet all of the requirements of the next iterator in the chain. For example, as a stringent test of both `Transpose` and our generic algorithms (and also of a C++ compiler's ability to handle templates), we can try composing `Transpose` with itself (continuing the above example):

```
   typedef Transpose<TransType, int, (ptrdiff_t)cols, (ptrdiff_t)rows>
           DoubleTransType;
   DoubleTransType l(k);
   quickSort(l, l + rows * cols, Comparator<int>());
```

This results in the array being sorted in row order, just as when we worked directly with the original `int*` iterator type.

4 Concluding remarks

An algorithm-oriented approach to generic software library development has been outlined and illustrated by a small sample of generic algorithms coded in C++. The basic approach is similar to that of our earlier work in Ada, but is adapted to the specific language features available in C++. We have also placed more emphasis than in the Ada work on describing implementation design decisions in the documentation. These design decisions arise both from known optimizations that carry over from concrete versions of the algorithms and from constraints imposed by the need to operate in a wide variety of contexts.

The form of the documentation used in this paper is only an approximation to what will probably be necessary. Some potential library users may find the degree of abstraction baffling or the amount of detail overwhelming. This problem can probably be best solved by structuring the documentation in several layers, beyond the two illustrated in this paper (overview of a collection of related algorithms and data sheets on individual algorithms). For example, another layer could be provided that specifies a "typical" concrete instance of each algorithm; a programmer inexperienced with the notion of algorithmic abstraction might find it useful to examine this layer first, then progress to the more general descriptions.

While we have opted for run-time efficiency by using strictly compile time mechanisms for instantiating parameters, one could instead emphasize run-time flexibility and reduction of code size by defining some of the access operations as virtual functions [4, p. 208] that are implemented in derived classes. Such a choice fits within our framework because it does not require any textual changes to the source code of the algorithms, only to the container classes.

In this paper, we have concentrated on issues of development and documentation of the individual algorithmic components, but we recognize there are other important aspects of the development and effective use of software libraries, which we plan to address in future reports.

Acknowledgments Meng Lee and Mehdi Jazayeri are also designers of the present library and worked on many of the components mentioned in the paper. We would like to thank them, Bob Cook, and two anonymous referees for many suggestions for improvement of the paper.

References

[1] G. Booch, *Software Components with Ada*, Benjamin/Cummings, 1987.

[2] G. Booch and M. Vilot, "The Design of the C++ Booch Components," *Proc. OOPSLA/ECOOP '90*, SIGPLAN Notices, Vol. 25, No. 10, October 1990.

[3] T. H. Cormen, C. E. Leiserson, R. L. Rivest, *Introduction to Algorithms*, McGraw-Hill, New York, 1990.

[4] M. Ellis and B. Stroustrup, *The Annotated C++ Reference Manual*, Addison-Wesley, New York, 1990.

[5] D. E. Knuth, *The Art of Computer Programming, Volume 3: Sorting and Searching*, Addison-Wesley, Reading, Mass., 1973.

[6] D. Lea, *The GNU C++ Library*, software and documentation, The Free Software Foundation, 675 Mass Ave, Cambridge, MA, Feb 1988.

[7] D. R. Musser and A. A. Stepanov, "A Library of Generic Algorithms in Ada," *Proc. of 1987 ACM SIGAda International Conference*, Boston, December, 1987.

[8] D. R. Musser and A. A. Stepanov, "Generic Programming," invited paper, in P. Gianni, Ed., *ISSAC '88 Symbolic and Algebraic Computation Proceedings, Lecture Notes in Computer Science 358*, Springer-Verlag, 1989.

[9] D. R. Musser and A. A. Stepanov, *The Ada Generic Library: Linear List Processing Packages*, Springer-Verlag, 1989.

[10] D. R. Musser and A. A. Stepanov, *Algorithm-Oriented Generic Software Library Development*, Rensselaer Polytechnic Institute Computer Science Department Technical Report 92-13, April 1992.

[11] R. Sedgewick, "Implementing quicksort programs," *Communications of the ACM*, 21(10):847-857, 1978.