SOFTWARE—PRACTICE AND EXPERIENCE, VOL. 24(7), 623–642 (JULY 1994)

Algorithm-oriented Generic Libraries

david r. musser
Rensselaer Polytechnic Institute, Computer Science Department, Troy, New York 12180, U.S.A. (email: musser@cs.rpi.edu)

and

alexander a. stepanov
Hewlett-Packard Laboratories, Software Technology Laboratory, 1501 Page Mill Road, Palo Alto, California 94303, U.S.A. (email: stepanov@hplabs.hp.com)

SUMMARY

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.

key words: Generic algorithms Algorithmic abstractions Software libraries Abstract data types C++ Templates

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. References 1 or 2) or preprocessing mechanisms (e.g. Reference 3), achieving greater run-time efficiency than was possible with older methods, such as 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

CCC 0038–0644/94/070623–20
Received 1 June 1992
Revised 17 February 1994

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 operations parameters, we permit a single expression of the algorithms to be used with any concrete representation of the container.

Outline of the algorithm-oriented approach

The key steps of our approach to generic library construction are

1. 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.
2. 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.
3. 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.
4. 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:

(a) selection of the algorithms to be used
(b) selection of a container representation suited to the selected algorithms and other constraints
(c) 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
(d) instantiating other parameters of the generic algorithms, such as data types and problem size.

Altogether, this flexibility means that 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.

The algorithm-oriented approach in C++

We demonstrated an earlier version of the algorithm-oriented approach in Ada in References 4–6. 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 an array representation of sequences or 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.

TYPE REQUIREMENTS ON ITERATORS AND APPLICATIVE OBJECTS

Nongeneric algorithms are expressed in terms of types that are fixed built-in or user-defined types, not subject to substitution. Generic algorithms are expressed in terms of type parameters that can be instantiated with any actual types that meet certain requirements. Part of the specification of a generic algorithm is the set of requirements on types. The fewer the requirements the more generic the algorithm is, since there are more possibilities for actual types with which the algorithm can be instantiated. We thus need good ways to specify the type requirements without overspecifying them. In C++, one can use the template mechanism to express a generic algorithm, as in

```
template 〈class T〉
void f(T x) {
  T y;
  … g(y);
}
```

Here T, though called a class, can be instantiated with any type, either one defined by the C++ class mechanism or a built-in type. Within the body of f, T can be used explicitly in declarations (as shown) or implicitly as the supplier of functions such as g. Although it could be made explicit that g must come from T by writing T::g(y), the other requirements one needs to place on a function such as g cannot be spelled out completely within the language. Although one can infer the number and types of the arguments and return value of g from calls such as … g(y), one might prefer to have a separate specification of these type requirements against which calls could be type-checked. Similarly, nothing about semantic or computing time requirements on g can be specified in C++, though in some cases some requirements might be deduced from requirements on the functionality and computing time of f.

Lacking facilities in C++, we must adopt some meta-level specification methodology for spelling out the requirements a generic algorithm places on its template type parameters. There are many possibilities, some (such as algebraic axioms or model-based specifications) more formal than others. In this paper our approach is essentially model-based, but not fully formal in that we do not spell out all the specifics of the main mathematical model used, finite sequences; instead we use standard terminology and notation for finite sequences and appeal to the reader's prior experience with these mathematical objects.

Two of the broad categories of types that are used in the algorithms of our library are *iterator* types and *applicative* types. Iterators generalize the notion of pointers, encapsulating information about locations in objects. Applicative types provide objects whose role is purely to supply the definition of one or more functions; they are useful for expressing higher-order algorithms, i.e. ones that take functions as parameters. Iterator and applicative types can be defined in C++ using the class mechanism; any such definition or built-in type that meets certain requirements, detailed below, will do. The requirements are all on attributes of operations: operator names, argument lists, meanings, and computing times.

Iterators

Iterators provide for sequencing through a sequence of locations and obtaining information from them. Generally, iterator types are defined in C++ using the class mechanism, but for an important class of iterators called 'random access' iterators (defined below) for sequence containers, they may simply be C++ pointer types, i.e. T* where T is any built-in or user-defined type). This is possible since the requirements we place on random access iterators are consistent with C++ notation and operation meanings for pointer types. Some of the general requirements for iterators, including all of the requirements that must be met by sequence iterator types, are spelled out in this section.†

Equality and inequality testing

An iterator object is said to refer to a *location* within a container; sometimes we also say it refers to the value, or 'element', in that location. We require that all iterator types provide

    int operator==(Iterator)   (equality check)

† The remainder of this section should be treated as a reference manual for iterator terminology and requirements; for a first reading those familiar with C++ may want to just skim this material and rely mainly on their knowledge of C++ pointer notation and semantics for understanding the algorithm descriptions in later sections. The ReverseIterator example of an iterator defined by a C++ class should also help to clarify the role of the iterator requirements.

int operator!=(Iterator)   (inequality check)

where the equality operator returns true (i.e. a nonzero value) if its operands refer to the same location, false (i.e. zero) otherwise; and the inequality operator has the opposite meaning. These are required to be constant time operations.

Dereferencing

An operation we require of all iterator types is dereferencing (operator*()). Related to dereferencing is the classification of iterators is as *readable* or *writable* (or both). If an iterator x for a container of elements of type T refers to location *i*, the meaning of *x is given as follows: If *i* is a valid location for dereferencing, then

1. If *x is in a context that requires a value of type T—in C++ parlance, an 'rvalue' is required, as for example on the right-hand side of an assignment—and x is a readable iterator, *x means the element at location *i*.
2. If *x is in a context that requires a location—in C++ parlance, an 'lvalue' is required, as for example on the left-hand side of an assignment—and x is a writable iterator, *x means location *i* itself.

Otherwise, *x is undefined.

In C++, a purely syntactic means of making an iterator readable but not writable is to declare its dereference operator as T operator*() or const T& operator*(). To make an iterator writable but not readable, one can make its dereference operator return the iterator itself and define an operator=(const T&) on the iterator.

A *regular* iterator is one that is both readable and writable; regular is the default if no mention is made of readability or writability. Other useful terms are *read-only*, meaning readable but not writable, and *write-only*, meaning writable but not readable.

Dereferencing is required to be a constant time operation.

Sequence elements and locations

For the remainder of this paper we restrict the discussion to the case that the container is a sequence. We assume that for any (finite) sequence of elements x_0, x_1, …, x_{n-1} there is a sequence of distinct locations i_0, i_1, …, 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. Locations i_0, …, i_{n-1} are valid for dereferencing, but i_n might or might not be.

An array representation of sequences commonly uses a set of integers in an arithmetic progression as locations, with the off-the-end location just being the next higher integer in the progression following the location of the last element, whereas a linked-list representation uses a set of pointers, usually with the null pointer as the off-the-end location.

Traversal classification of sequence iterators

An important classification of sequence iterators is based on the kinds of traversal they efficiently support: *forward*, *bidirectional*, and *random access*. Loosely speaking, forward iterators are those that provide for efficient traversal through the locations

$i_0, i_1, \ldots, i_n$ in that order; a bidirectional iterator is a forward iterator that also provides for efficient traversal in the opposite direction; and a random access iterator is a bidirectional iterator that also provides for efficient 'long jumps' in the sequence and for comparisons based on relative position in the sequence. The precise definitions of forward, bidirectional, and random access iterators are given in terms of the names and meanings of the operations they are required to efficiently support, as detailed below.

Ordering of locations in a sequence

We treat the locations $i_0, i_1, \ldots, i_n$ of a sequence as being ordered by their indices $0, 1, \ldots, n$. Each location i_{j+1} is called the *successor* of i_j, and i_j is called the *predecessor* of i_{j+1}, for $j = 0, \ldots, n-1$.

We say that location i_j is *before* location i_k if $j < k$ and is *after* location i_h if $h < j$. Thus it makes sense to speak of the *first* location in a sequence with a particular property: no location before it has that property.

The notion of 'before' is defined for all sequence iterators but forward and bidirectional iterators do not necessarily provide any efficient way of computing it. For random access iterators, we do require a constant-time comparison operation `int operator<(Iterator)` which returns true if its first operand refers to a location before that referenced by its second operand; i.e., if `x` refers to i_j and `y` refers to i_k, then `x < y` returns true (i.e. a nonzero value) when $0 \le j < k \le n$, false (zero) when $0 \le k \le j \le n$, undefined otherwise. Random access iterators for sequences must also provide other constant-time comparison operators `<=`, `>`, and `>=`, whose meanings are defined similarly.

Successor and predecessor operations

All sequence iterators must also provide a traversal operation (`operator++()`) for advancing from a location to its successor; bidirectional iterators must also provide 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,[7] we define the requirements on both:

1. For $j = 0, \ldots, n-1$, if the iterator `x` refers to location i_j then `x++` or `++x` causes it to refer to i_{j+1}; `x++` returns the original iterator that refers to i_j whereas `++x` returns the new iterator that refers to i_{j+1}. If `x` refers to i_n, then the effect and return values of both `x++` or `++x` are undefined.
2. For $j = 1, \ldots, n$, if the iterator `x` refers to location i_j then `x--` or `--x` causes it to refer to i_{j-1}; `x--` returns the original iterator that refers to i_j while `--x` returns the new iterator that refers to i_{j-1}. If `x` refers to 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.

Random access operations

A random access sequence iterator must provide

```
Iterator operator+(ptrdiff_t)        (addition)
Iterator operator−(ptrdiff_t)        (subtraction)
```

with the meaning that for $j = 0, \ldots, n$, if x refers to location i_j, then if $0 \leq k \leq n - j$ then the iterator returned by x + k refers to i_{j+k}, and if $0 \leq k \leq j$ then the iterator returned by x − k refers to i_{j-k}. Both operations must be constant time operations.

A random access sequence iterator must also provide 'long jump' operators

```
Iterator operator+=(ptrdiff_t)       (positive long jump)
Iterator operator−=(ptrdiff_t)       (negative long jump)
```

with the meaning that for $j = 0, \ldots, n$, if x refers to location i_j then if $0 \leq k \leq n - j$ then x is changed by x += k to refer to i_{j+k}, and if $0 \leq k \leq j$ then x is changed by x −= k to refer to i_{j-k}. In both cases the resulting iterator is returned. Both operations must be constant time operations.

It is sometimes useful to construct forward or bidirectional iterator types that provide operations that have these names and meanings but that do not meet the requirement of constant time execution. For example a linked-list representation only permits a positive long jump to be programmed with (the equivalent of) iteration of ++, and thus it is a linear time operation in that case. When combined with algorithms that expect constant time long jumps, execution speed is degraded, but in some cases not by much if other parts of the computation make significantly larger contributions to the total time. For example, our library contains a binary search algorithm that can be combined with a forward iterator for a linked-list representation, producing an algorithm that is linear in the number of iterator operations but only logarithmic in the number of comparisons it does. In many cases in which comparisons are more expensive than iterator operations, such an algorithm can beat a straight sequential search, which is linear in both iterator operations and comparisons.

Iterator subtraction

Finally, some algorithms require a sequence iterator to provide

```
int operator−(Iterator)      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 $j - k$.† A random access iterator must provide iterator subtraction as a constant time operation. A forward or bidirectional iterator is not required to provide iterator subtraction, but if it does the operation must execute in linear time.

Iterator ranges

In describing algorithms that use sequence iterators, it is convenient to use range notation. Let i_j and i_k be two locations in the same sequence, with $j \leq k$; one or both of i_j or i_k might be the off-the-end location. Then

† Note that this is defined even if x or y holds i_n, the 'off-the-end' value; for example, if x holds i_n and y holds i_0 then x − y returns n, the length of the sequence.

range $[i_j, i_k]$	is $i_j, i_{j+1}, \ldots, i_k$
range $[i_j, i_k)$	is $i_j, i_{j+1}, \ldots, i_{k-1}$
range $(i_j, i_k]$	is $i_{j+1}, \ldots, i_k$
range (i_j, i_k)	is $i_{j+1}, \ldots, i_{k-1}$

If x is an iterator that refers to i_j and y is an iterator that refers to i_k, then [x, y] means the same as $[i_j, i_k]$ and similarly for the other kinds of ranges. Many of the sequence algorithms take iterator parameters first and last, which are regarded by the algorithm as specifying the range [first,last), i.e. the location referred to by last is not regarded as part of the sequence that is processed by the algorithm. Note that in the case that first = last, the range [first,last) is empty; for such an input most algorithms would do nothing or would return a default value.

The forward iterator requirements are met by a singly-linked list class in our library. This class also provides addition, positive long jumps, and iterator subtraction, but in linear rather than constant time. The bidirectional iterator requirements are met by a doubly-linked list class; linear-time subtraction and negative long jumps are also provided.‡ All random access iterator requirements are fulfilled by pointer types in C++. The ReverseIterator type given in a later section is another example of a random access iterator.

Applicative objects

Some of the algorithms in our library 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(), and to create types that provide such an overloading. We call such types *applicative* types.

For example, our sorting algorithms are parameterized by a *comparator* type, i.e. an applicative type that provides 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.

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. Using the constructor, intComparator, we can create an object of this class and pass it to a function, such as the quickSort function described in a later section, with a call such as

‡ 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.

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

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. The constructor call for T = int would then be as in

```
quickSort(first, last, Comparator⟨int⟩());
```

In general, we say that a type is an *applicative type* if it is defined by a C++ class that provides one or more definitions of the function call operator. Since a C++ compiler can inline the definition of the function at the site of calls, using applicative 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!

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 and some auxiliary data movement algorithms. Some of these algorithms require bidirectional iterators; others require random access iterators. The partitioning and sorting algorithms also require a comparator type to define the function used to compare elements of the sequence.

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 and follow it with datasheets for individual algorithms.

Overview

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 $O(n^2)$ worst-case computing time on a sequence 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. The partitioning algorithm used by quickSort, and by other algorithms, is unguardedPartition, which permutes a sequence into two subsequences, one containing elements that compare less than or equal to a given value, and the other containing elements that compare greater than or equal to the value.

Concrete versions of these algorithms may be found in standard references, e.g. References 8 and 9, and the research literature, e.g. Reference 10. 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 just abandon the sentinel technique and provide 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.

Algorithm datasheets

Insert an element into a sorted range

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).

Type requirements. Iterator must be a regular bidirectional iterator.

Group. Unary pseudo-permutation.

Time complexity. Linear. The number of T assignments is the size of the range from the insertion point to last.

Space complexity. Constant.

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

unguardedLinearInsert assumes there is some location before last that holds a value no larger than value; if the last such value is in location *p*, it inserts value in location *p* after shifting the values in the range [*p*,last) over by one location. If the sequence in locations [*p*,last) was previously in ascending order according to compare, then the resulting sequence in the range [*p*,last] is also in ascending order according to compare.

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

Sort a range by insertions (insertion sort)

Declaration.

```
template ⟨class Iterator, class Comparator⟩
void insertionSort(Iterator first, Iterator last,
    Comparator compare);

template ⟨class Iterator, class Comparator⟩
void unguardedInsertionSort(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 range [first,last) 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 [first,last) occurs in the first threshold locations. These functions are not recommended for general use but are a good choice for sorting short or 'almost sorted' sequences.

Type requirements. Iterator must be a regular random access iterator.

Group. Unary pseudo-permutation.

Time complexity. Quadratic, in the average and worst cases. The number of compare operations performed is about $n^2/4$ in the average case and about $n^2/2$ in the worst case, and the number of T assignment operations is the same, where n is the size of [first,last). For most inputs these functions are very slow compared to the best sorting algorithms. However, they 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. For such sequences the worst case time is linear in the size of the sequence.

Space complexity. Constant.

Details. All these functions are stable sorts; that is, the relative order of elements that are equal (according to compare) is preserved.

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 range [p,first) is already sorted and the value in location p is a minimum for the *extended* range [p, last). The result is that [p,last] is sorted into ascending order. Note that if $p \neq$ first, the sequence unguardedInsertionSort leaves in [first,last] is in ascending order but is not a permutation of the values originally in those locations (some values change places with those in [p,first)).

Implementation.

```
template ⟨class Iterator, class Comparator⟩
void insertionSort(Iterator first, Iterator last,
    Comparator compare)
{
    if (first == 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)
{
    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 beginning to end 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 preceding ones, shifting elements over 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 beginning location, first. Hence it depends on the assumptions stated in ‘Details’ 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 ‘Details’ are satisfied for the call to unguardedInsertionSort.

Partition a range

Declaration.

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

Description. Permutes the range [first,last) in place, partitioning it into two ranges such that compare $(*i,\text{pivot}) \leq 0$ for all locations i in the first range and compare $(*j,\text{pivot}) \geq 0$ for all locations j in the second range. Returns an iterator that refers to the beginning location of the second range.

Type requirements. Iterator must be a regular random access iterator.

Group. Unary permutation.

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

Space complexity. Constant.

Details. There must be at least one location i for which compare$(*i,\text{pivot}) \leq 0$ and at least one location j for which compare$(*j,\text{pivot}) \geq 0$. These conditions are met if there is at least one location i in [first,last) for which

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

The beginning location p of the second range is in [first,last]. (Thus, either subsequence may be empty.) Unlike some versions of partitioning, it is not guaranteed that

compare(*p,pivot) $= 0$

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.)

Implementation.

```
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 beginning for an element that compares non-negative with pivot, search from the end 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 inner loops need no check for running off the end of the sequence: by the assumption described in 'Details', 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.

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.

Sort a range by partitioning (quicksort)

Declaration.

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

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 it can be unacceptably slow.

Type requirements. Iterator must be a regular random access iterator.

Group. Unary permutation.

Time of complexity. Order $n \log n$, on the average, where n is the size of [first,last). 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).

Details. This is 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 first part are less than or equal to all of the elements in the second 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. 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.

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.

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

The final insertion sorting takes only linear time, since no element is more than t 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 t locations.

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 $\log_2 n$.

There can be up to $n - t$ partitionings, on sequences of length $n, n-1, \ldots, t+1$, 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.

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.

Datasheets for two other functions used in implementing quicksort, moveBackward and medianOf3Select, may be found in Reference 11.

AN EXAMPLE OF A SEQUENCE ITERATOR TYPE

The generic algorithms discussed in the previous section can be used not only with the built-in C++ pointer types for the iterators, but with any user-defined type that ments all of the requirements of a random access iterator. In this section, we present a simple example of such an iterator type, one that provides for traversal in the reverse direction from that defined by a given iterator type.

Overview

To allow our algorithms to work with the sequence of elements in reverse order, the ReverseIterator class definition transforms a given iterator into one for which, for example, ++ has the meaning of the original iterator's −−, and vice versa. As an example of the use of this iterator type, consider sorting with a combination of quickSort, ReverseIterator, and a comparator that would ordinarily produce ascending order:

```
const size_t length = 100;
void main() {
    int a[length];
    ...
    typedef ReverseIterator⟨int*⟩ ReverseInt;
    ReverseInt k(a + length);
    quickSort(k, k + length, Comparator⟨int⟩());
```

where the template class Comparator is as defined earlier. Note that k refers to a[length−1] and k + length refers to an off-the-end location (k + length − 1 refers to a[0]). Since the sequence is being scanned in the reverse of the normal order, the result produced is sorted into ascending order when scanned in reverse order, and thus is in descending order when scanned in the normal order.

Iterator datasheet

Reverse the direction of an iterator

Declaration.

```
template ⟨class Iterator, class T⟩
class ReverseIterator;
```

Description. From a given random access iterator type, Iterator, this class defines a new random access iterator type that reverses the direction of Iterator's traversal.

Type requirements. Iterator must be a regular random access iterator.

Provides. ReverseIterator provides a regular random access iterator type.

Time complexity. All operations are constant time, provided that all Iterator operations are constant time.

Space complexity. Constant.

Details. If [first,last) is a range of size n defined for Iterator, a declaration of the form

```
ReverseIterator⟨Iterator⟩ i(last);
```

sets up i to refer to last − 1 and to traverse a range whose locations are last − 1, last − 2, ..., first in that order, followed by an off-the-end location. Computing the off-the-end location does not require location first − 1 to be defined for Iterator.

Implementation.

```
template ⟨class Iterator, class T⟩
class ReverseIterator {
protected:
    Iterator current;
public:
    ReverseIterator(Iterator x) : current(x) {}
    T& operator*() const {return *(current − 1);}
    int operator==(ReverseIterator⟨Iterator, T⟩& other) const
        {return current == other.current;}
    int operator!=(ReverseIterator⟨Iterator, T⟩& other) const
      {return current != other.current;}
    int operator<=(ReverseIterator⟨Iterator, T⟩& other) const
      {return other.current <= current;}
    // ... similar definitions for <, >, and >=.
    ReverseIterator⟨Iterator, T⟩ operator++() {current−−; return *this;}
    ReverseIterator⟨Iterator, T⟩ operator−−() {current++; return *this;}
    ReverseIterator⟨Iterator, T⟩ operator++(int)
      {ReverseIterator⟨Iterator, T⟩ tmp = *this; current−−; return tmp;}
    ReverseIterator⟨Iterator, T⟩ operator−−(int)
      {ReverseIterator⟨Interator, T⟩ tmp = *this; current++; return tmp;}
    ReverseIterator⟨Iterator, T⟩ operator+=(ptrdiff_t k)
      {current −= k; return *this;}
    ReverseIterator⟨Iterator, T⟩ operator−=(ptrdiff_t k)
      {current += k; return *this;}
    ReverseIterator⟨Iterator, T⟩ operator+(ptrdiff_t k) const
      {ReverseIterator⟨Iterator, T⟩ tmp = *this; return tmp += k;}
    ReverseIterator⟨Iterator, T⟩ operator−(ptrdiff_t k) const
      {ReverseIterator⟨Iterator, T⟩ tmp = *this; return tmp −= k;}
    ptrdiff_t operator−(ReverseIterator⟨Iterator, T⟩& other) const
      {return other.current − current;}
};
```

Implementation notes. The class maintains an Iterator location in current and uses it to compute the next new location requested. Dereferencing is applied to current − 1 so that a reverse iterator initialized to the last location for a range [first,last) can traverse all the locations of the range in reverse order and use first as the off-the-end location for the new range.

Iterator and applicative type transformers

ReverseIterator 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 ReverseIterator and our generic algorithms—and also of a C++ compiler's ability to handle templates—we can try composing ReverseIterator with itself:

```
const size_t length = 100;
void main() {
    int a[length];
    ...
    typedef ReverseIterator⟨int*⟩ ReverseInt;
    ReverseInt k(a + length);
    typedef ReverseIterator⟨ReverseInt⟩ DoubleReverseInt;
    DoubleReverseInt l(k + length);
    quickSort(l, l + length, Comparator⟨int⟩());
```

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

Similarly, we can define applicative type transformers. One example would be a comparator transformer that inverts the comparison, providing another way of changing an ascending sort into a descending one. As another example, consider

```
template ⟨class BinaryFun, class T⟩
class IndirectBinaryFun {
  BinaryFun b;
public:
  IndirectBinaryFun() {}
  operator()(T* x, T* y) {return b(*x, *y);}
};
```

which transforms any binary function type to one that uses a level of indirection. By thus transforming a comparator type and combining it with quickSort, we immediately obtain a version of quickSort that can work with an array of pointers to the actual values and thus only move the pointers, not the actual values. Such a version is of course particularly useful for sorting large records. Conventionally the source code of such a version would have the indirection done inline and thus it would have to be distinct from the normal version, but with our approach the adaptation is done by the compiler and only the one version of the source code has to be maintained.

These are but a few of the many cases in which several different useful versions of the same algorithm are obtainable from a single generic algorithm by combining with different iterators or comparators.

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 best be solved by structuring the documentation in several layers, beyond the two illustrated in this paper, overviews 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.

Although 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 (Reference 7, 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 that there are other important aspects of the development and effective use of software libraries, which we plan to address in future papers.

acknowledgments

Meng Lee is also a designer of the present library and worked on many of the components mentioned in the paper. We would like to thank her, Bob Cook, Mehdi Jazayeri, 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,* **25**, (10), 1–11 (1990).
3. D. Lea, *The GNU C++ Library,* software and documentation, The Free Software Foundation, 675 Mass Ave, Cambridge, MA, February 1988.
4. D. R. Musser and A. A. Stepanov, 'A library of generic algorithms in Ada', *Proc. 1987 ACM SIGAda International Conference,* Boston, December 1987, pp. 216–225.
5. 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.
6. D. R. Musser and A. A. Stepanov, *The Ada Generic Library: Linear List Processing Packages,* Springer-Verlag, 1989.
7. M. Ellis and B. Stroustrup, *The Annotated C++ Reference Manual,* Addison-Wesley, New York 1990.
8. T. H. Cormen, C. E. Leiserson and R. L. Rivest, *Introduction to Algorithms,* McGraw-Hill, New York, 1990.
9. D. E. Knuth, *The Art of Computer Programming, Volume 3: Sorting and Searching,* Addison-Wesley, Reading, Mass., 1973.
10. R. Sedgewick, 'Implementing quicksort programs', *Communications of the ACM,* **21**, (10), 847–857 (1978).
11. D. R. Musser and A. A. Stepanov, 'Algorithm-oriented generic libraries', Rensselaer Polytechnic Institute Computer Science Department, *Technical Report 93-23,* September 1993, revised January 1994.