Complete Traversals and their Implementation Using the Standard Template Library

Eric Gamess(*), *David R. Musser*(†) and *Arturo J. Sánchez-Ruíz*(*)

(*) Laboratorio de Construcción de Herramientas Automáticas (AuTooLab)
Centro de Ingeniería de Software y Sistemas (ISYS)
Escuela de Computación, Facultad de Ciencias
Universidad Central de Venezuela
APARTADO 47642, CARACAS 1041–A
VENEZUELA
E–mail: {*egamess*, *asanchez*}*@anubis.ciens.ucv.ve*
http://*anubis.ciens.ucv.ve*/∼*asanchez*/*autoolab.html*

(†) Computer Science Department
Rensselaer Polytechnic Institute
Troy, NY 12180
USA
E–mail: *musser@cs.rpi.edu*
http://*www.cs.rpi.edu*/∼*musser*

Abstract

A *complete traversal* of a container S (such as a set) is defined by the iteration scheme

for all $x \in S$
$\mathcal{F}(x, S)$

where $\mathcal{F}$ is a function that might possibly modify S by inserting new elements. We assume that the order in which the elements are treated is not relevant, as long as the iteration continues until $\mathcal{F}$ has been applied to all elements currently in S, including those $\mathcal{F}$ has inserted. Standard iteration mechanisms, such as the iterators provided in the C++ Standard Template Library (STL), do not directly support complete traversals. In this paper we present two approaches to complete traversals, both extending the STL framework, one by means of generic algorithms and the other by means of a container adaptor.

Keywords: Generic Programming, Standard Template Library (STL), Iterators, Adaptors, Containers, Templates, C++.

1 Introduction

Consider the following problem:

> A manager wants to arrange a meeting of a certain set of people in her company. For each person in the original set she also wants to invite that person's boss, that boss's boss, and so on. (She has a database from which she can tell who a person's boss is.)

The manager can solve this problem fairly simply by writing down the initial set of people's names in a list and iterating through the list from beginning to end, inserting new persons at the end where they become part of the iteration. To avoid duplicating names on the list, she should append a

person's name to the list if and only if the name is not already present. In the computerized version of this problem, with a large list, an inefficient linear search is required, rather than the binary search that would be possible if the set of names could be kept in, say, alphabetic order. In that case, however, new names should be inserted in their proper place to maintain the order. But this in turn makes it difficult to tell when the iteration should stop, since names might have been inserted before the current iteration point. The kind of iteration required to gracefully solve this problem is called a *complete traversal*; we give a formal definition in Section 2. Problems requiring complete traversals are fairly common (we give another example in Section 2), and while there are various *ad hoc* ways of solving them, programmers should ideally have at their command an efficient packaged solution. In this paper we describe two such approaches to complete traversal, both of which fit into the framework defined by the Standard Template Library, STL (part of the ANSI/ISO draft standard for C++ [2]).

STL [8, 5, 1] provides a set of easily configurable software components of six major kinds: generic algorithms, containers, iterators, function objects, adaptors, and allocators. In each of these component categories, STL provides a relatively small set of fundamental components; it is through uniformity of interfaces and orthogonality of component structure that STL provides functionality far beyond the actual number of components included. But STL is not intended as a closed system; its structure is designed with extension in mind. The complete traversal components described in this paper may be of interest not only for the functionality they provide, but also as examples of, and measures of, how well the existing STL components support extensions.

In Section 3, we give two distinct ways of solving the complete traversal problem: a generic algorithms approach and a container adaptor approach. In both approaches, the complete traversal components are designed to work with the category of STL components called *associative containers*, which support fast retrieval of objects based on keys. The generic algorithms are restricted to *sorted* associative containers, in which keys are maintained according to a given ordering function, but the container adaptor we provide can also be used with *hashed* associative containers, which give up order properties in favor of faster retrieval. Hashed associative containers are not part of the draft C++ standard but are now provided as an STL extension by at least one compiler vendor [1]. Another classification of associative containers is *unique*, in which objects in a container cannot have equivalent keys, versus *multiple*, in which they can. Still another classification is *simple* containers, in which only the keys are stored, versus *pair* containers, in which pairs of keys and associated values are kept. The sorted associative containers provided in STL are shown in the following table:

Component	Classification
`set<Key, Compare, Allocator>`	unique, simple
`multiset<Key, Compare, Allocator>`	multiple, simple
`map<Key, T, Compare, Allocator>`	unique, pair
`multimap<Key, T, Compare, Allocator>`	multiple, pair

All of the associative containers have essentially the same interface; e.g., each provides `insert` and `erase` member functions for inserting and deleting objects, several kinds of search member functions, and several kinds of iterators for traversing through the current contents. None, however, provides for complete traversals in the sense discussed here. The specifics of these interfaces, and how our components are used for complete traversals, are illustrated at the end of Section 3, in terms of solving the manager's problem stated at the beginning of the paper.

We give more than one approach to the complete traversal problem because no single solution seems best in all cases. The presentation in Section 3 includes complexity analyses and discussion

of other factors such as naturalness of interfaces. We are exploring still other approaches, which are discussed briefly in the concluding section.

2 Complete Traversals

We begin this section by giving a precise definition of complete traversals. We then describe another example application of the concept.

The definition is dependent on whether the given container S is unique (i.e., there are no repeated elements, as in an STL set or map) or multiple (i.e., repeated elements are allowed, as in an STL multiset or multimap). In the unique case, for a given function $\mathcal{F}$, let $\hat{\mathcal{F}}(x, S)$ be the set of elements inserted in S by $\mathcal{F}(x, S)$. Then the iteration scheme

$$\begin{array}{c} \textbf{for all } x \in S \\ \mathcal{F}(x, S) \end{array}$$

is called a *complete traversal* of S if $\mathcal{F}$ is applied to exactly the elements in the set $\mathcal{S}$ defined by

$$\begin{aligned} \mathcal{S}_0 &= S \\ \mathcal{S}_{i+1} &= \bigcup_{x \in \mathcal{S}_i} \hat{\mathcal{F}}(x, \mathcal{S}_i), \quad i = 0, 1, \ldots \\ \mathcal{S} &= \bigcup_{i=0}^{\infty} \mathcal{S}_i \end{aligned}$$

If S is a multiple container, then $\hat{\mathcal{F}}(x, S)$ is defined as a multiset, the unions in the definition are replaced by multiset unions, and $\uplus_{x \in S}$ means each x is repeated the number of times it occurs in S.

In the case that S is unique, it follows from the definition that the complete traversal is finite if and only if there is some k such that $\mathcal{S}_{k+1} \subseteq \bigcup_{i=0}^{k} \mathcal{S}_i$. In the multiple case, we have the stronger requirement that $\mathcal{S}_k$ is empty for all sufficiently large k. In what follows we assume complete traversals are finite. Although a finite complete traversal could be computed by actually constructing the sets or multisets $\mathcal{S}_i$, we seek solutions that are more space and time efficient.

As described in Section 1, the manager's invitation list is one problem that might solved by instantiating a complete traversal scheme. We show in detail such a solution using our complete traversal components in Section 3.4. As another example, the one which inspired this study of complete traversals, suppose we are given a specification of types written in a certain formalism called CTS.[1] CTS types are classified into *atomic types* (which have no structure), *concrete types* (which are used to define data structures), and *abstract types* (which are used to define classes). The representations associated with classes are, in turn, concrete CTS types. With each CTS expression we can associate a *syntax graph*, which captures the relationships among involved types. We also assume that each CTS expression defining a type is associated with a name. Given a map S of elements (n, g), where n is the name of an abstract CTS type and g is the syntax graph associated with its representation, we want to iterate over S in such a way that, at each iteration, we take an element (n, g) and traverse g in a certain order, making sure that we insert into S elements (n', g'), for all type names n' we come across, where g' is the syntax graph associated with n'. Iteration stops when all elements in S have been processed. This instance of the complete traversal problem is summarized in Fig. 1, where S is shown with its initial value.

[1] CTS stands for Common Type System. The application problem and the definition of CTS were taken from [7].

$S = \{(n, g) \mid n$ *is the name of an abstract* CTS *type and* g *is the syntax graph of its representation*$\}$

$\mathcal{F}(x, S) =$ *traverse* $x.g$ *and insert in* S *any type name found with its corresponding syntax graph*

Figure 1: An instance of the complete traversal problem

3 Complete Traversals Implemented as STL Extensions

In this section we present two different approaches to complete traversals, one using generic algorithms and the other a container adaptor. We also compare the complexity of these components, and show how they can be used in a simple application.

3.1 Generic Algorithms

A generic algorithm is an algorithm designed to work with a variety of data structures, the specialization to a particular structure being realized by the programming language processor (compiler, interpreter, or run-time system) rather than by manual editing of the source text. In the STL framework, generic algorithms are expressed as C++ function templates. An algorithm can be made generic over a category of containers if the way it accesses a container can be limited to a fixed set of operations, all of which are provided, with the same interfaces, by every container in the category. As a simple example, consider the following function template for performing an (ordinary) iteration over the elements of a container, applying a function `f` to each element.

```
template <class Container, class Function>
void for_each(Container& container, Function f)
{
  Container::iterator i;
  for (i = container.begin(); i != container.end(); ++i)
    f(*i);
}
```

This function can be applied to any of the STL containers, because they all provide iterator types (`Container::iterator`) and member functions `begin` and `end` that return iterators defining the range of positions of elements currently within the container. To use `for_each` with a list of integers, for example, one could write

```
list<int> list1;
// ... code to insert some elements in list1
for_each(list1, f);
```

where `f` is some function object that does not modify the list.

The main algorithmic idea behind our first approach is to set up an iteration through the container with the ordinary iterators provided, applying a function `f` that may generate new elements for insertion into the container. But instead of allowing `f` to do the insertions, we require it to enter the new elements in a queue. After each call of `f`, we take elements from the queue and insert them

into the container, checking whether they are inserted before or after the current iteration position. If an element is inserted after the current position, it will be taken care of in the course of the remaining iterations, but if it is inserted before the current position, we apply `f` to it immediately (which in turn may generate new elements and add them to the queue).

STL defines several different categories of iterators according to the set of operations they support, the most powerful being random access iterators. While random access iterators support relational operators, the bidirectional iterators provided by associative containers do not. However, we can still easily tell whether an element x appears before another element y in the iteration sequence of a sorted associative container, just by comparing x with y using the order relation $\prec$ the container uses to maintain sorted order.

There are then two distinct cases, depending on whether the container is unique or multiple. A unique container does not allow two equivalent elements to be in the container. Equivalence of elements is defined as x is equivalent to y if both $x \prec y$ and $y \prec x$ are false. With a unique container, if `f` generates an element equivalent to one already present (either before or after the insertion point) we do not apply `f` to it again, but with a multiple container we do.

For the unique container case, we define the generic algorithm `complete_unique_traversal`, which traverses a unique sorted associative container applying a function object `f` to each element. It is assumed that `f` creates and maintains a queue of elements to be inserted (but does not insert them itself), and that it makes this queue available as a public member named `Q`. When inserting an element `v` taken from this queue into the container, we use an insertion function that returns a pair `p` consisting of an iterator that tells where the element was found or inserted, and a boolean value that is true if and only if the element was actually inserted (was not already present):

```
pair<UniqueSortedAssociativeContainer::iterator, bool>
  p = container.insert(v);
```

The boolean value then is given by `p.second`. The full definition of `complete_unique_traversal` is as follows.

```
template <class UniqueSortedAssociativeContainer, class Function>
void complete_unique_traversal(UniqueSortedAssociativeContainer& container,
                               Function f)
{ UniqueSortedAssociativeContainer::value_type v;
  UniqueSortedAssociativeContainer::iterator i;
  for (i = container.begin(); i != container.end(); ++i) {
    f(*i, container);
    while (!f.Q.empty()) {
      v = f.Q.front();
      f.Q.pop();
      pair<UniqueSortedAssociativeContainer::iterator, bool>
        p = container.insert(v);
      if (p.second && container.value_comp()(v, *i))
        // v has been inserted in container (it wasn't already there)
        // and it occurs before the current traversal
        // position, i, so process it now with f:
        f(v, container);
    }
  }
}
```

For the multiple container case, the generic algorithm `complete_multiple_traversal` is defined similarly,[2] but the implementation is complicated by the fact that when an element taken from the queue is equivalent to the one at the current position, testing for whether it is inserted before or after that position is no longer simply a matter of comparing it with the current element. The draft C++ standard specification of the `insert` operation on multiple containers leaves unspecified where within a range of equivalent elements a new equivalent element is inserted, so we must conduct a linear search within the range. Using the original Hewlett-Packard implementation of STL, we could omit the linear search since an element is always inserted at the end of the range of elements equivalent to it, but we cannot assume this to be the case with all implementations since the standard does not require it. This situation is a simple illustration of the tension faced by the library (or language) specifier, between the goal of allowing implementors as much freedom as possible by leaving some details unspecified, and the goal of enabling programmers to optimize their code while retaining portability.

3.2 A Container Adaptor

The implementation based on generic algorithms, shown in Section 3.1, requires function `f` (implemented by the programmer) to put the elements generated in each activation in a queue, which is less natural than having it insert the elements directly into the container. In order to relax this requirement, we propose another approach based on a container adaptor, whose usage for implementing complete traversals is depicted in Fig. 2. From a given container, the programmer builds a

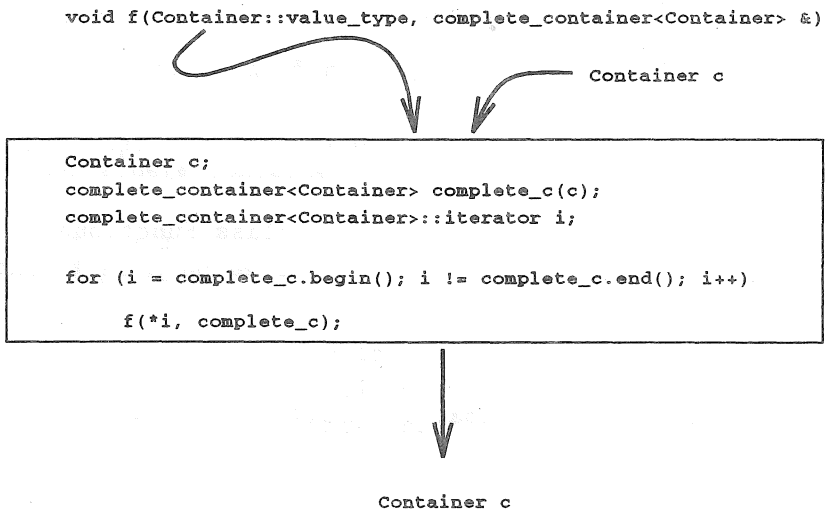

Figure 2: Implementing complete traversals by using a container adaptor approach

`complete_container` whose representation consists of a reference to the input container, plus a data structure needed to implement complete traversals. The `complete_container` adaptor[2] provides:

- Types `size_type` and `value_type` taken from the corresponding `Container`.
- A constructor that takes a `Container` as a parameter and stores a reference to its argument, and also creates an iteration list (see below).

[2]The implementation is available from `ftp://kanaima.ciens.ucv.ve/pub/autoolab/stl`.

- Types `iterator` and `const_iterator`, which implement complete traversal iterators on non-constant or constant containers, respectively.
- Member function `size`, which returns the size of the `Container` component of a `complete_container`.
- Member function `insert` which takes a `value_type` value and inserts it into the underlying `Container`.

The representation of a `complete_container` consists of a container reference and an iteration list, implemented as an STL `list<value_type>`. The implementation maintains the following invariant:

> *At the beginning of each iteration with an* `iterator`*, the element associated with the iterator on the iteration list is the one to be processed, the elements to its right are those that will be processed next, and elements to its left are those which were already processed.*

This invariant implies that, initially, the iteration list should be a copy of the original `Container`, and insertion of an element with the adaptor's `insert` member should insert the element into the underlying container and onto the end the iteration list.

3.3 Complexity

To compare the time complexity of our two approaches, let C be a sorted associative container, let n be the number of elements in C initially and let m be the total number of insertions done by the complete traversal of C using a function f. There might be fewer than $N = n + m$ elements in C after the traversal because some of the elements being inserted might have already been present. But N is a bound on the final size of C, so $O(\log N)$ bounds the time for any one insertion, and $O(m \log N)$ bounds the time for all insertions. Let $T(f, j, k)$ be a bound on the total amount of time for j evaluations of f on a container of maximum size k, where we exclude (because we have already counted it) any time f spends doing insertions. So the total time for evaluating f is $T(f, N, N)$.

1. In `complete_unique_traversal`, the time for all of the queue processing is $O(m)$, so the total time is

$$\begin{array}{rccccc} & \text{queue processing time} & + & \text{insertion time} & + & \text{function evaluation time} \\ = & O(m) & + & O(m \log N) & + & T(f, N, N) \end{array}$$

   The extra linear searches required by `complete_multiple_traversal` add to these times an extra $O(mN)$ term in the worst case, but in practice the extra time is likely to be negligible.

2. For the `complete_container` adaptor, the time for all of the list processing is $O(N)$, so the total time is

$$\begin{array}{rccccc} & \text{list processing time} & + & \text{insertion time} & + & \text{function evaluation time} \\ = & O(N) & + & O(m \log N) & + & T(f, N, N) \end{array}$$

Since $T(f, N, N)$ is $\Omega(N)$, the bound in 1 cannot be asymptotically better than the bound in 2. It is clear, however, that the list processing time associated with complete containers is more than the queue processing associated with the complete traversal algorithms. It is also clear that the complete traversal algorithms require less extra space than the complete containers. On the other hand, complete containers offer a more natural interface and can be used with hashed associative containers, while the same cannot be said of our complete traversal algorithms. In summary, these two approaches offer a good spectrum of possibilities to tackle the complete traversal of containers.

3.4 An Example

As a simple example of use of the generic components described in this section, we program the solution of the manager's invitation list problem described in Section 1. First we present a solution using one of the generic algorithms of Section 3.1.

```
// Read a bosses database file, another file containing an initial
// set of persons, and compute a complete traversal of the set, inserting
// as a new member the boss of anyone already present.
#include <iostream.h>
// ... other #includes, for STL and string class headers
#include "complete_traversal.h"  // contains complete_unique_traversal algorithm

// A type of map from strings to strings, alphabetically ordered:
typedef map<string, string, less<string> > name_association;

// A type of set of strings, ordered by alphabetic ordering of the keys:
typedef set<string, less<string> > name_set;

// A class of function objects for generating names using a name_association
// directory, meeting requirements of the complete_unique_traversal algorithm:
class name_function {
private:
  const name_association& directory;
public:
  name_function(const name_association& d) : directory(d) { }
  queue<list<string> > Q;
  void operator()(const string& name, name_set& s) {
    cout << name << endl;
    name_association::const_iterator i = directory.find(name);
    if (i != directory.end())
      Q.push((*i).second);
  }
};

// Function to scan the database file and build an internal directory
void get_database(istream& is, name_association& directory); // details omitted

// Function to scan the names file and build a names set
void get_names(istream& is, name_set& names); // details omitted

int main()
{ // Create the bosses database:
  name_association bosses;  ifstream ifs("bosses.txt");
  get_database(ifs, bosses);

  // Create the initial set of names:
  name_set invitees;  ifstream ifs1("initial.txt");
```

```
  get_names(ifs1, invitees);

  cout << "Original set of invitees:" << endl;
  for (name_set::iterator i = invitees.begin(); i != invitees.end(); ++i)
    cout << *i << endl;

  cout << "Output during complete traversal:" << endl;
  name_function get_boss(bosses);
  complete_unique_traversal(invitees, get_boss);

  cout << "Final set of invitees:" << endl;
  for (name_set::iterator i = invitees.begin(); i != invitees.end(); ++i)
    cout << *i << endl;
}
```

A solution using the `complete_container` adaptor of Section 3.2 needs a different definition of the function to be applied to each person, one that actually does the insertions. Assuming this function object is called `insert_boss`, the adaptor could be used as follows:

```
  typedef complete_container<name_set> cc_type;
  cc_type cc(invitees);
  for (cc_type::iterator k = cc.begin(); k != cc.end(); ++k)
    insert_boss(*k, cc);
```

4 Related Work

CLU [3] is one of the earliest contributions which offers language support for defining iterators as operations on programmer-defined container types. Since the programmer has total control over how iteration is defined, supporting complete traversal would be possible, perhaps by adapting one of the approaches discussed here. In [3], the authors mention the potential usefulness of such iterators but develop neither a formal definition nor any examples.

More recently, the work reported in [4] on list iterators in C++ covers issues associated with iterator integrity; i.e., problems which may arise when the object to which an iterator is pointing is deleted. Even though this work does not deal with complete traversals, the iterator integrity problem would come into play if we tried to do complete traversals on STL sequence containers (e.g., vectors or deques), because insertion in vectors and deques might require memory reallocation which invalidates all iterators pointing to the container in question. Except for the case of such iterator invalidation, complete traversals of STL sequence containers can be trivially programmed.

Another recent related work is the Java Generic Library (JGL) [6], which is strongly based on the STL design. For instance, JGL supports the concept of containers and iterators. However, it does not appear that complete traversals are directly supported.

5 Summary and Future Work

We have defined the complete traversal of a given container S as an iteration scheme which consists of iterating over S applying, at each iteration, a function $\mathcal{F}(x, S)$ which might possibly modify S

by inserting new elements into it. The iteration should stop when all elements currently in S have been processed.

In order to offer packaged solutions to programmers who need to use this class of iteration schemes, we have presented two approaches to perform complete traversals implemented on the platform provided by STL. Our first approach is in terms of generic algorithms that require that the iterated function create a queue to hold the generated elements. One algorithm handles containers with unique keys, while the other handles containers with multiple keys. By taking special precautions in the case of multiple keys, we remain independent of the details of particular implementations of sorted associative containers.

Our second approach is based on a complete container adaptor. The main features of this adaptor are the special iterators and insert operation it provides, by which the programmer can obtain complete traversals using a function that directly inserts new elements in the container.

The time complexities of both our approaches are asymptotically equivalent. However, the approach based on generic algorithms stores just the elements which are generated at each iteration in the function's queue, while the container adaptor stores all container elements in its iteration list. On the other hand, our complete container can be used with any STL associative container (including existing extensions such as hashed containers and any future extensions meeting the requirements of associative containers), while the generic algorithms can be used with sorted associative containers only.

There are still other approaches we are exploring, such as the result of melding the complete container idea with the approach of keeping just the generated elements. We are also interested in trying iteration schemes which do deletions as well as insertions. In this direction we have conjectured the nonexistence of functions $\mathcal{F}$ which do insertions and deletions and are such that the order of traversal is irrelevant. The connection between complete traversals and iterator integrity is also on our list of future work. We are also exploring the relationship between complete traversals and what we call *iterator trajectory functions*; i.e., functions object which describe a specific way of traversing a set. Lastly, we plan to use the components presented in this paper to solve real-life applications, like the CTS application mentioned in Section 2, and to measure the performance of both approaches with randomly-generated containers.

References

[1] Silicon Graphics. Standard Template Library Programmer's Guide. URL http://www.sgi.com/Technology/STL, 1997.

[2] Accredited Standards Committee X3 Information Processing Systems Doc No: X3J16/96-0225 WG21/N1043. *Working Paper for Draft Proposed International Standard for Information Systems Programming Language C++.* American National Standards Institute (ANSI), 1996.

[3] B. Liskov and J. Guttag. *Abstraction and Specification in Program Development.* MIT Press, 1986.

[4] H. J. Messerschmidt. List iterators in C++. SOFTWARE—PRACTICE AND EXPERIENCE, 26(11):1197–1203, 1996.

[5] D. R. Musser and A. Saini. *STL Tutorial and Reference Guide.* Addison-Wesley, 1996.

[6] ObjectSpace. Objectspace – JGL, the Generic Collection Library for Java. URL http://www.objectspace.com/jgl/jgl_documentation.html, 1997.

[7] Arturo J. Sánchez Ruíz. *On Automatic Approaches to Multi–Language Programming via Code Reusability.* Phd dissertation, Computer Science Department, Rensselaer Polytechnic Institute, Troy, NY, USA, 1995.

[8] A. A. Stepanov and M. Lee. The standard template library. Technical Report HP-94-93, Hewlett–Packard, 1995.