Charles: A Data Structure Library for Ada95

Matthew Heaney

On2 Technologies, Inc.
mheaney@on2.com
http://home.earthlink.net/~matthewjheaney/index.html

Abstract. Charles is a container library for Ada95, modelled closely on the C++ STL[1–4]. Sequence containers (vectors, deques, and lists) store unordered elements, inserted at specified positions. Associative containers (sets and maps) order elements according to a key associated with each element; both sorted (tree-based) and hashed containers are provided. A separate iterator type[5, 6] associated with each container is used to visit container items and to allow elements to be modified directly. Charles is flexible and efficient, and its design has been guided by the philosophy that a library should stay out of the programmer's way. Charles is for getting your work done.

1 Introduction

In general, the Charles library is designed using static mechanisms (generic packages and subprograms) rather than dynamic ones (tagged types). "Static polymorphism" is an important programming idea that is often undervalued. A set of loosely-related but otherwise disparate types each have an identical interface, and you compile against this "abstract" interface. By programming to the interface instead of to a specific type, then it becomes relatively easy to change the type behind the interface.

An important goal in the design of libraries is that abstractions should be safe and easy to use. Therefore, in Charles memory management for (unbounded) containers is automatic, by implementing container types as a private derivation of type `Controlled`. The fact that container types are (privately) tagged also means that equality composes, so it is not possible for predefined equality to ever reemerge (satisfying this constraint is necessary because the library is deliberately designed to facilitate the composition of one component from another).

However, in library design there is often a tension between flexibility and safety. In general, where there has been conflict between these goals, in Charles it has been resolved in favor of flexibility. This does not imply that there should be no safety at all, but rather that it is best effected by the client himself, via some mechanism *outside of* the library, at a higher lever of abstraction. Systems are built from the bottom up.

Another philosophy in the design of Charles is that it is *easy* to do common things, and *possible* to do less-common things (perhaps not as easily, but possible

nonetheless). For example, creating a container type whose elements are non-limited and definite (the common case) requires only a single instantiation (the common case is easy). Generic formal subprograms always have defaults ("`is <>`"), which simplifies instantiations (especially for generic algorithms) because matching actuals directly visible at the point of instantiation do not need to be passed explicitly. Container types are non-limited, because this makes it easier to compose abstractions (and not because we need assignment, which can be effected via other mechanisms). In general the library is designed so that a container type (or its associated iterator type) can be used directly as the generic actual element type of another container instantiation.

One property of a good user interface is that a user should be able to guess what something does, and his guess should always be right. Therefore type names and coding style closely follow long-established RM conventions. In particular, operation and parameter names were largely inspired by the `Unbounded_String` package. For type names Charles follows the convention of the predefined I/O packages (each of which use `File_Type` as the name of the abstraction) and uses the name `Container_Type` for all of its containers. The symmetry with files is deliberate, because it emphasizes that files are essentially persistent containers.

Note that the actual container types are (publicly) declared as traditional abstract data types, not as tagged types. A theme of the library is that you create a complex abstraction by composing generic components, not by extending a tagged type. It is not necessary for the library to provide dynamic polymorphism for containers, because a client can add that himself using a thin layer (a "type adapter"[5]) of glue code.[1]

A separate type — an *iterator* — is associated with each container. An iterator is a mechanism for manipulating elements in a container. There are operations for navigating among elements, and for selection and modification of the element currently designated by the iterator. An important feature of this schema is that it allows you to view the collection as a just a "sequence of elements," thus abstracting-away the actual container. This is valuable because now a generic algorithm (for sorting, say) can be written in terms of just an iterator, so that you can use the algorithm over any container having an iterator with the requisite operations. Iterators also obviate the need for special container conversion operations, e.g. that return an array object with all the container elements. The iterator mechanism is sufficiently general that you can convert from any container type (including arrays) to any other container type (including arrays).

2 Sequence Containers

The elements in a sequence container are unordered, and may be inserted at specific positions. The family of sequence containers comprises vectors and queues,

[1] For this reason I specifically rejected the approach used the by Weller/Wright Ada95 Booch components[7], which implement containers as a family of tagged types.

for constant-time random access of elements, and lists, for efficient insertions and deletions in the middle of the sequence.

Charles does not have "stack" or "queue" containers because you can achieve the same effect by simply pushing and popping items from the appropriate end of one of the more general sequence containers. If a container is needed that, say, disallows access to the elements in the middle of the sequence, then the user can implement that himself as a thin layer on top of a primitive component.

2.1 Vectors

The canonical container is the vector, which is simply a linear sequence of items. A vector allows random access of elements (with only *unit* time complexity), and is optimized for insertions at the back end of container. For example:

```
V : Vector_Subtype;
Push_Back (V, New_Item => X);
```

An unbounded vector is implemented as a pointer to an internal array, that automatically grows as necessary to accommodate new items. When more storage is necessary, a new array is allocated that is two times the length of the existing array. (Array growth is always a function of internal size rather than the number of active elements in the vector. If it were otherwise, then insertion complexity would be quadratic rather than linear.)

Insertion at other positions within the vector is permitted:

```
Insert (V, Before => I, New_Item => X);
```

However, the time complexity is linear, because all the elements from the insertion position to the back of the vector have to slide up to make room for the new item. This is why there is no `Push_Front` operation for vectors (although the effect of such an operation can achieved by `Insert`'ing an item before the position `Index_Type'First`).

The key benefit of a vector is that it supports random access of elements in constant time:

```
X := Element (V, Index => I);
Replace_Element (V, Index => J, By => Y);
```

Here we have a selector function that returns the value of the element at the indicated position. However, returning a copy of the element is not a sufficiently general mechanism, as efficiency issues might prohibit copying. And what about modifying the element itself? Consider a hash table implemented as vector of lists – how does a client add an item to the list at a certain index position?

To satisfy this need, all the containers in Charles have an additional generic selector function that accepts an access type as a generic formal type, and which returns an access value designating the element. This gives us a variable-view of the actual element object, which we can then manipulate directly:

```
   function To_Access is new Generic_Element (List_Access);
   List : List_Subtype renames To_Access (V, Index => I).all;
begin
   Push_Back (List, New_Item => X);
```

Although a vector automatically resizes as necessary during insertion, if you know the ultimate size of the vector *a priori* then it's always more efficient to perform the allocation in advance. The vector has factory functions for constructing the container at the time of its declaration:

```
   procedure Op (N : Natural) is
      V : Vector_Subtype := To_Container (Length => N);
   begin
```

There is also a `Resize` operation to explicitly specify the size of the internal array.

2.2 Deques

A "deque" (short for **d**ouble-**e**nded **que**ue) also supports random access of elements, but unlike a vector insertion of an element at the front of the container has only *constant* time complexity. It is implemented as an unbounded array of pointers to fixed-size blocks, with each page comprising a fixed number of elements. `Push_Front` works by adding new elements to the first page (working from back to front on the page), and decrementing an offset that keeps track of which element on the page is the first element of the container. When there is no more room on the front page, then a new page is allocated and the offset is reset. The internal array expands as necessary to address all the internal pages. `Pop_Front` works similarly, except that it increments the offset.

As with a vector, the deque allows elements to be inserted (deleted) at any position, but the time complexity is still linear. One difference from a vector is that to make room for the new element, the existing elements are moved toward the end of the deque that is closest to the insertion position.

2.3 Lists

Lists are optimized for insertions at any position. All insertions and deletions, in the middle or at either end, have constant time complexity. However, random access is not supported, and advancing from one position to another has a time complexity proportional to the distance between positions.

Rather than referring to elements using a discrete index (as for vectors and deques), for lists we use a separate iterator type:

```
   I : constant Iterator_Type :=  Last (List);
   E : Element_Type renames To_Access (I).all;
```

There are dedicated operations for pushing an item onto the front or back of the list. To insert an item at a certain position, we specify the position using an iterator value:

```
   Insert (List, Before => Iterator, New_Item => Item);
```

The find operation returns an iterator value, which we can test against the distinguished Back iterator value to determine whether the search was successful:

```
   I : constant Iterator_Type := Find (List, Item);
begin
   if I /= Back (List) then ... --search succeeded
```

If you need to move elements from one list onto another, a list is optimal because it doesn't require any copying. Instead, only pointers need to be manipulated (the internal node of storage is moved from one list to another, and the respective element counts are adjusted accordingly). For example, calling Splice like this:

```
   Target, Source : List_Subtype;
   I, J : Iterator_Type;
   ...
   Splice (Target, I, Source, J);
```

moves (*not* copies) the item designated by iterator J in list container Source, onto the list Target just prior to the element designated by iterator I. Additional overloadings of Splice generalize this to move an entire range of elements.

The list container also provides operations for reversing the list, sorting the elements, removing duplicates, filtering, and for merging two lists. The sort operation is stable (the relative order among equivalent items is preserved across a sort), and works by exchanging pointers to nodes, not by copying elements. For example:

```
   function "<" (L, R : ET) return Boolean is ...;
   function Sort is new Generic_Quicksort; --"<" is default
   function Is_Same (E1, E2, : ET) return Boolean is ...;
   function Purge_Duplicates is new Generic_Unique (Is_Same);
begin
   Sort (List);
   Purge_Duplicates (List);
```

The library provides both singly-linked and doubly-linked list containers. Both forward and reverse iteration over a double list is provided, but only forward iteration is possible for single lists. However, if all you need is forward iteration, then the singly-linked list is more space efficient because the overhead per node is smaller. The single list also caches a pointer to the last node, so that pushing an item onto the back of the list can be performed in constant time. (It would be linear otherwise, because it would necessary to traverse the list from the front, just to locate the last item.[4])

Note that the list containers are "monolithic" data structures, meaning that there is no structure sharing (except what is implied by the use of iterators). List assignment is by value, not by reference. The intent of the name "list" is to emphasize that this container is optimized for constant-time insertion and deletion of elements at arbitrary positions. The semantics of this abstraction are *not* the same as for the similarly-named "polylithic"[6] data structure in functional languages as LISP.

3 Associative Containers

In addition to sequence containers, the Charles library has "associative" containers that associate a "key" with each element, and which order the keys for efficient retrieval. For a set container an element is its own key, but a map container allows you to index an item by some other arbitrary type (in this sense a map is a generalization of an array).

Both hashed and sorted associative containers are provided. A hashed container has possibly linear time complexity in the worst case, but the average time complexity is *constant*. A sorted container guarantees *logarithmic* time complexity even in the worst case.

In Charles the sorted containers are implemented as red-black trees[8], but this does not preclude other choices — an AVL tree could have been used just as easily. Indeed, multiple implementations would be possible. However, all the components have an identical[2] container interface (because of static polymorphism), so once a component (whether sorted or hashed) has been instantiated, the implementation more or less disappears.

A hashed associative container orders keys by comparing hash values for equality. Keys that hash to the same value are stored in the same bucket, which is implemented as a singly-linked list. (This implies that reverse iteration is not available for hashed containers.) Keys for a sorted associated container are compared using an *equivalence* relation, not by equality. This is called "strict weak ordering." You can think of equivalence as being implemented this way:

```
function Is_Equivalent (L, R : Key_Type) return Boolean is
begin
   return not (L < R) and then not (R < L);
end;
```

To store an item in a sorted associative container the client must provide an order relation for the keys. The canonical example would be a set of employee records, ordered by social security number:

[2] Strictly speaking they aren't really identical, because there are operations that make sense for some implementations, but not others. For example, a hashed container has a `Resize` operation to specify the size of the hash table, but this obviously wouldn't be applicable for a tree-based implementation.

```
   type Employee_Type is
      record
         SSN : SSN_Type;
         ...
      end record;

   function "<" (L, R : Employee_Type) return Boolean is
   begin
      return L.SSN < R.SSN;
   end;

   package Employee_Sets is
      new Charles.Sets.Sorted.Unbounded (Employee_Type, "<");
```

Now you can add a new employee to the database like this:

```
   E : Employee_Type := ...;
begin
   Insert (Set, New_Item => E);
```

and it will be inserted according to the value of `E.SSN`. If we want to change some information for an employee (let's say his address changed), then we can use `Find` (really, an instantiation of `Generic_Find`), which returns an iterator designating the employee record object:

```
   I : constant Iterator_Type := Find (S, Key => SSN);
   E : Employee_Type renames To_Access (I).all;
begin
   E.Address := New_Address;
```

There is only a subtle difference in how sets and maps are implemented. For a set, the element is its own key, and the set doesn't allocate space for anything other than the element. For a map, the key is separate piece of data, and the map is implemented as key/element pairs. In either case storage is ordered according to the key (per the relational operator for sorted containers, or the hash value for hashed containers).

Note that equality for maps is implemented in terms of the equality operator for elements; keys do not participate in determining whether two maps are equal. Set equality is also defined in terms of the element equality rather than the relational operator used to compute equivalence.

In order to store a key in a map, it must have a definite subtype. One issue is that some keys are more naturally represented using indefinite subtypes, e.g. a name having type String. To use the map the client would first have to convert the key to the definite subtype used to instantiate the package, and then call the map operation. However, this is neither syntactically graceful, nor very efficient (temporaries are created only to be immediately destroyed). For these reasons, and because maps with string keys are nearly ubiquitous, the library provides dedicated map abstractions having type String as the key. The string-key maps

also accept a key comparison operation as a generic formal subprogram; for example this would allow key comparisons to be case-insensitive.

Except for `Storage_Error`, the associative containers do not raise exceptions. The search operation, for example, does *not* raise an exception if the key was not found. Rather, the value of the `Back` iterator is returned as a nonce, to indicate failure. For example:

```
      I : constant Iterator_Type := Find (Map, Key);
   begin
      if I /= Back (Map) then ...
```

A similar approach is used for insertions into unique-key maps, and in particular insertion of a key into a map that already contains the key *does* ***not*** *raise an exception*. Rather, a Boolean value is returned indicating whether the insertion was successful:

```
declare
   Iterator : Iterator_Type;
   Success  : Boolean;
begin
   Insert (Map, Key, Item, Iterator, Success);
   if Success then ...
```

If Key is already in the Map, then Success returns False, and the iterator designates the key/element pair whose key matches Key. Otherwise Success returns True and the iterator designates the newly-inserted pair.

Conditional insertion is a necessary feature of any efficient[3] map abstraction. Consider an application that counts the frequency of words in a text file, implemented as a map (the histogram) with key type String and element type Natural. The algorithm can be implemented by simply always trying to (conditionally) insert a count of 0 for a word:

```
procedure Insert (Word : String) is
   Iterator : Iterator_Type;
   Success  : Boolean;
begin
   Insert (Histogram, Word, 0, Iterator, Success);

   declare
      Count : Natural renames To_Access (Iterator).all;
   begin
      Count := Count + 1;
   end;
end Insert;
```

[3] It's efficient because the test to determine whether the key is already in the map, and the insertion if it isn't, is an atomic operation, and therefore the tree traversal is only done once.

Here we don't care about the whether the insertion was successful; our interest is only that the insertion be *conditional*. If the word is already in the map, the insertion fails, and we simply increment its current count value. If the word was not already in the map, the insertion succeeds, and the associated count is initialized to the value 0 (and then immediately incremented).

For many applications an unconditional replacement of the existing value is appropriate, analogous to replacing the value of a component of an array. The associative containers overload the `Insert` operation with versions that either create a new element if the key does not exist or simply replace the current element if it does.

Multiset and multimap containers allow multiple keys to be equivalent. Therefore there is no conditional version of Insert because all insertions are *unconditional*. Unless memory is exhausted, insertions always succeed, by allocating a new key/element pair in the map. Elements with equivalent keys are stored in adjacent positions, as a contiguous range. To interrogate elements whose keys are the "same," operations return a pair of iterators designating the ends of the range. For example, suppose we want to look up all the people whose last name is `"Heaney"`:

```
   Iter, Back : Iterator_Type;
begin
   Equal_Range (People_Map, "Heaney", Iter, Back);

   while Iter /= Back loop ...
```

Here, `Equal_Range` returns an iterator pair specifying a half-open range of elements; this permits iteration in the normal way. The `Lower_Bound` and `Upper_Bound` operations can be used to return just the respective endpoint.

It is a necessary feature of any general-purpose container library that it provide a way to modify container elements in-place. However, for an associative container, we must take special care to ensure that a key is not modified accidently. This would break the abstraction, which depends on key values being in equivalence order. For a set, the `Generic_Element` operation imports the access type as `access constant` rather than `access all`, so it returns a constant view of the object instead of a variable view, and thus prevents the set element from being modified. The `Generic_Key` operation for the map similarly returns only a constant view. (Note that map correctness only depends on keys, not elements, so its `Generic_Element` operation doesn't need any special treatment.)

However, we accept that some applications have a legitimate need to modify set elements. In our employee set example, we modified the element by changing the address component of the employee record. That was a perfectly safe (and reasonable) change, because we didn't modify the key. To accommodate this need, the set container provides a special operation, `Generic_Modify_Element`, that imports the generic formal access type as `access all` in the normal way, and which therefore returns a variable view. The spelling is deliberately different, so that if the container instantiation is changed to a set from some other

component, then any elements that had been modified via `Generic_Element` (as for other containers) would be immediately flagged at compile time. The function `To_Access` that we showed in the example was an instantiation of the special operation. There is a corresponding operation for the map container, `Generic_Modify_Key`, to admit benign changes to key values.

If you do need to change a key, you *must* delete the element from the container first, make the change and then re-insert the element. For example:

```
   I : Iterator_Type := Find (Set, Key => SSN);
   E : Employee_Type := Element (I); --make a copy
begin
   Delete (Set, Iterator => I);
   E.SSN := New_SSN;
   Insert (Set, New_Item => E);
```

The hashed associative containers import a hash function as a generic formal subprogram. Charles provides hash functions for common types as String and Integer. The containers are implemented as an unbounded array of buckets (the hash table), with each array component a linked list of elements. The array itself always has a length that is a prime number, which produces a better scatter when `mod`'ing the hash value.

The hash table automatically resizes itself as items are inserted, in order to maintain a load factor of 1; this is how it can guarantee that average time complexity is constant. When the load factor is exceeded, a bigger array is allocated and the existing elements are rehashed onto the new hash table. This means that occasionally (whenever the array needs to grow) there will be spikes in the execution time of insert.[4] However, as with a vector, there is a resize operation that allows you to preallocate the buckets array to a specified size.

4 Iterators

Containers are important to the extent that we care how quickly we can find an element in the container, or how efficient it is to insert an element, but ultimately it is the *elements* in the container in which we are interested. An iterator is a structured way of gaining access to the elements, without exposing the representation of the container.

A reusable abstraction should be as flexible, efficient, and safe as possible, but these goals are often in conflict. This tension is particularly acute in the design of iterators, because they essentially violate the encapsulation provided by the container in order to allow access to its elements. How type safe should an iterator be? Should a container keep track of all the iterators currently designating items

[4] It's not clear whether this would be acceptable in real-time programs, which demand predictable behavior and must be designed around worse case execution times. There is another technique called "incremental hashing" that can be used to smooth out the spikes, and I'm currently debating whether to implement hash tables this way instead.

in the container? What should happen if you try to remove an item from the container while it's being designated by an iterator? Charles doesn't try to solve these problems, as that would have grossly compromised the design. Where there has been tension among these goals, I have in general traded type safety for flexibility and efficiency. A client can always build a safe layer on top of a flexible and efficient abstraction, but the opposite is not true.

I prefer to program close to the machine, and therefore iterators have been closely modelled on raw access types. In particular, to be sufficiently general and flexible the iterator type *must* be non-limited and definite. To discover what iterator operations are needed, we can examine how access types are used to traverse a simple linked list:

```
declare
   Node : Node_Access := List.Head;
begin
   while Node /= null loop
      Process (Node.Element);
      Node := Node.Next;
   end loop;
end;
```

From this example we can extrapolate what operations need to be provided by a high-level container:

- An operation to return an iterator designating the "first" element.
- An operation to return a sentinel, to which the iterator is compared to determine whether it has "fallen off the end" of the sequence.
- An operation to inspect the "current" element designated by the iterator.
- An operation to return an iterator designating the "next" element.

If we implement an iterator as a thin wrapper around an access value, this allows us to hide the representation of the container, but preserves the flexibility and efficiency of raw access types. We can then rewrite the example like this:

```
declare
   I : Iterator_Type := First (List);
   B : constant Iterator_Type := Back (List);
begin
   while I /= B loop
      Process (Element (I)); --or Process (To_Access (I).all)
      I := Succ (I);
   end loop;
end;
```

This schema can be generalized to work for any container besides a list, and this is in fact how iterators are implemented in Charles. The fact that the iterator is non-limited and definite means you can implement any other type in terms of

the iterator. This is important because real systems are built from the bottom up, by assembling complex abstractions from simpler primitives.

The library has been designed to be flexible, so it is easy to compose abstractions (one container can have another container as its element type), and this is no less true for iterators (so you to can have a container of iterators, too). Consider a set, which orders its elements according to the relation specified when the set package was instantiated. Suppose we want to sort the elements according to some other criterion? One way would be to copy the elements into a list, and then sort the list. However, if the elements are large, or there are a large number of elements, efficiency issues might prohibit this approach. A more efficient technique would be to create a list of set iterators, that designate the elements in the original set, rather than creating a list of set elements. (This assumes, of course, that set iterators are small compared to set elements.) You could then sort the list of iterators according to whatever criterion you desire, irrespective of how the original elements are ordered in the set.

5 Conclusion

Charles satisfies the need for a general-purpose container library. It tries to find the right balance between flexibility and ease of use, and resolve the tension among the orthogonal goals of efficiency, safety, and flexibility. Writing complex abstractions that have even modest container needs is a lot simpler with Charles than without.

References

1. Plauger, P.J., Stepanov, A.A., et al.: The C++ Standard Template Library. Prentice Hall PTR (2001)
2. Musser, D.R., Derge, G.J., Saini, A.: STL Tutorial and Reference Guide. 2nd edn. Addison-Wesley Publishing Company (2001)
3. Stroustrup, B.: The C++ Programming Language. 3rd edn. Addison Wesley Longman, Inc. (2000)
4. Austern, M.H.: Generic Programming and the STL. Addison Wesley Longman, Inc. (1999)
5. Gamma, E., et al.: Design Patterns. Addison-Wesley Publishing Company (1995)
6. Booch, G.: Software Components With Ada. The Benjamin/Cummings Publishing Company, Inc. (1987)
7. Weller, D., Wright, S.: Ada95 Booch Components. World Wide Web, `http://www.pushface.org/` (1995)
8. Cormen, T.H., Leiserson, C.E., Rivest, R.L.: Introduction To Algorithms. The MIT Press (1990)