Charles: A Data Structure Library for Ada95

Matthew Heaney

April 30, 2002

Charles is a data structure library for Ada95, modelled principally on the C++ STL. It features both ordered (lists and arrays) and unordered (sets and maps) collections.

Associated with each data structure type is a separate iterator type, which allows you to visit each item in the container. In particular, an iterator abstracts away differences in specific container types, allowing you to view the collection simply as a sequence of items. A generic algorithm (for sorting, say) can be written in terms of an iterator, so that you can use the algorithm over any data structure having an iterator with the requisite operations.

Data structures are categorized by their time and space semantics. A list has constant time insertion and removal of items, but searching for an item requires linear time. An array ("vector") has constant time searching ("random access"), but insertions and removal require time proportional the length of the array. (A "deque" is another STL data structure that supports constant time insertion of items at the either end of the sequence, and allows random access of elements. This data structure is planned for a future version of Charles.)

In addition to lists and vectors, the Charles library has set, multi-set, map, and multi-map data structure types. A map is an "associative container" that allows you to index an item by some other arbitrary type. These data structures are extremely useful for testing membership of an item in a collection. In Charles all of these types are are implemented using a red-black tree, so finding an item requires only logarithmic time. (Note that even though sets and maps are traditionally "unordered" data structures, for pragmatic reasons the item type used to instantiate the component must have a relational operator.)

Lists and arrays are ordered sequences, and you're allowed to insert an item at a specific position in the container. There are special operations for inserting an item in the front or back of the container. Insertion of an item into a set or map is performed according to its order relative to the items already in the container.

Charles doesn't have Stack or Queue container types specifically, because the ordered container types allow you to achieve the same effect by simply pushing and popping an item from the appropriate end. One data structure that is missing, however, is a priority queue; this is planned for a future version of the library. (You can use a set or map as a priority queue, but this is not as efficient, because a height-balanced tree requires more maintenance during insertions and removals.)

One of my design goals for Charles was that instantiation of components should be as painless as possible. My philosophy is that common things should be easy to do (and consequently that less common things should be less easy to do). The library has therefore been optimized for container items whose type is non-limited and definite. In general, creating a container type from a Charles component requires only a single instantiation, passing only the item type as a generic actual parameter.

Container types are declared as traditional abstract data types, not as tagged types. A theme of the library is that you create a complex abstraction by composing instantiations of generic components, not by extending a tagged type. (If polymorphism is required, then the client can use the Adapter pattern to provide that himself. For that reason I specifically

rejected the approach used by the Booch components, which implements containers as a hierarchy of tagged types, requiring two separate instantiations.)

It is useful to have set elements and map indices that are indefinite (a symbol table, for example, is a map whose index type is a string). One solution is to simply require that clients use a definite type directly (type Unbounded_String, say) when instantiating the data structure. However, in the absence of automatic type conversion (as you have in C++), manipulation of such a data structure would be awkward. Therefore, given the importance of unconstrained array types in general, and type String in particular, in Charles I have provided additional set and map components that accept an indefinite generic formal type.

The container types in Charles have both bounded (stack-based) and unbounded (heap-based) forms. For the bounded forms I pass the size as a discriminant of the type. (I prefer to do it this way rather than passing the size as a generic formal constant.) The unbounded forms have a generic formal pool object (of type Root_Storage_Pool'Class), which is used for all internal allocation. Charles provides a pool object to simplify instantiation for those clients that don't have specific storage pool needs.

There is the orthogonal issue of whether items in the collection can be limited. On the one hand, we desire the library be as general as possible, and therefore we should support collections of limited items (to implement a queue of File_Type, for example). On the other hand, this would require passing in a generic formal subprogram to assign items (because data structure types are themselves non-limited), which is in conflict with our goal to make instantiation as painless as possible. (A queue of integer items should require only that the integer type be passed as the generic actual type.)

My solution in Charles was therefore to have different versions of the data structure types, one for non-limited items and another for limited items. For limited items, the data structure type is itself limited. A operation for adding an item to the container doesn't actually accept an item argument; rather, it just creates a new "slot" for the item. A separate selector function returns a pointer to the item, so that the client can manipulate the actual item (not just a copy). This allows the component to be agnostic about whether the item is "really" limited. (Indeed, even for a type that has assignment, if the items are "large" it may be more efficient to store them as "limited" items in a limited container, as this prevents undesirable copying of items.)

When I began writing the library, I was unsure about whether to implement unbounded data structure types as controlled. One argument against is that you can add controlled-ness as necessary, at the point of declaration of the object. For example, you can implement a generic controlled helper type that calls a generic formal subprogram during its own finalization. You can use an access discriminant to bind the controlled helper object to the data structure object, and have it call an appropriate finalization operation. Another technique is that if the uncontrolled data structure is a component of some higher-level abstraction that is itself controlled, then the data structure can be manually finalized during finalization of the enclosing record.

However, in practice it's too easy to forget to finalize the data structure. It's one more thing to have to think about, and more likely than not you won't think about it. So it turns out to be easier to simply have the component automatically finalize itself. Another reason is that if data structures weren't controlled, then they'd have to be limited, in order to prevent structure sharing. But that would make it harder to create complex data structures comprising elements that are themselves simpler data structures.

The data structure types in the STL require separate iterator types for forward and backward traversal. This is because templates in C++ have the sense of type-safe macros: the text of the template is simply expanded inline, so the operations of the type have to be identically named. (For example, the increment operator ("++iter") moves forward for a forward iterator type, but backwards for a reverse iterator type.) In Ada, however, there is less coupling between the generic formal type and the generic actual type, and therefore only one iterator type is necessary. The only requirement is that the signature of an operation match, not the name. (That being said, it's awfully convenient that the name does match, because

this greatly simplifies instantiation when formal operations are marked having a default. Charles always defaults all generic formal subprograms.)

My goal was to make iterators as simple, general, and efficient as possible, and that generic algorithms be easy to instantiate. This means passing only an iterator type during the instantiation, and taking advantage of "default" generic formal subprograms to implicitly pass operations directly visible at the point of instantiation.

There is lots of debate about how type-safe iterators should be. Should a container keep track of the iterators designating items in the container? What should happen if you try to remove an item from the container while it's being designated by an iterator? However, worrying about this would have complicated the design, and made everything else much less efficient. (In general, in the design of Charles I have been willing to trade type-safety for flexibility and efficiency.)

For unbounded forms, iterator implementation is relatively simple, and usually it's just a thin wrapper around an access type that designates a node of internal storage. I implemented iterator types for bounded forms exactly the same as for the unbounded forms, but unfortunately this means I lost the accessability checks that Ada provides when manipulating pointers to locally declared objects. I am not entirely satisfied with that choice, and it may very likely change in a future version of the library.

(For example, an iterator for a bounded form could be implemented as an integer array index, and iterator operations could accept both the data structure and the iterator. The operation would be implemented by using the integer index to de-reference an array component. This means the signature for iterator operations of a bounded form would be different from an unbounded form, which means you wouldn't be able to use a generic algorithm directly. However, you could work around that by declaring local subprograms, that are implemented by calling the operations of the locally declared bounded data structure.)

It was also my goal that generic algorithms (such as a sort) also work directly on Ada array types, as is the case in C++. This can be effected in Charles by declaring local subprograms that look like iterator operations, and which operate on the locally declared array object. These subprograms are then implicitly passed as default actual subprograms to the instantiation of the algorithm, which is instantiated with the array index type as the "iterator" type.

In order to iterate over a range of items of a sequence, it is necessary to have a nonce item that acts as a sentinel to terminate the iteration. We refer to this as a "half-open" range, meaning that it includes the first element but not the last element. Unfortunately, this idiom conflicts with Ada's higher-level approach, which uses a "closed range" that includes both the first and last elements. (In an earlier version the library I tried an iterator design that used a closed range, but there were certain cases that caused dangling iterator references. I have therefore concluded that the current half-open design is superior.) One consequence of the current design is that different sentinels are needed for forward versus backwards iteration over a half-open range, so the data structures in Charles have selectors that return "first" and "back" iterators for forward iteration, and "last" and "front" iterators for backward iteration.

Charles also includes an "access control" type, which is exactly analogous the the auto_ptr type in C++. This is extremely useful for those times when it's necessary to manipulate access objects directly (polymorphic programming, for example). The abstraction reifies the notion of the "owner" of an access object who can transfer ownership to another control object, or release ownership entirely. If the access control object still owns an access object at the time of its finalization, then it automatically frees the designated object.