Ada Structured Library
Release 1.0

Corey Minyard (minyard@acm.org)

February 11, 1999

Contents

Preface

This document describes the Ada Structured Library (ASL), a set of library routines, frameworks, and container classes for Ada95. The document describes the following parts of the library:

- The Baseclass packages.
- The container classes
- The general library routines

Chapter 1

Overview of the Baseclass package

This package contains the base class used by the rest of the system. All tagged types should decend from the abstract types defined here.

Two abstract types are defined: Object and Limited_Object. Obviously, Object is not limited and Limited_Object is. No other differences exist.

The base classes each provide three methods:

To_String - Return a string representation for an object.

Get_Object_Name - Return a name for an object. For debugging errors, it is often useful if the object prints its actual name so the exact tagged type is known. All tagged types should define one of these functions.

Cast - A casting operator to allow interfaces and multiple inheritance.

1.1 The Cast Method

Ada95 does not include any direct support for multiple inheritance or interfaces. However, such support can be easily added to the system through consistent use of the Cast method.

The base Cast method simple returns the object pointer passed in. If the developer wants to create a tagged type with interfaces, they should create a new tagged type for

each interface or subclass. Then each interface should override the Cast method and provide the ability to convert between the various subclasses and interfaces. It should also call the parent Cast method for anything it doesn't want to implicitly handle.

For instance, suppose we have the following interface:

```
with Baseclass;

package Intf1 is

   type F1 is abstract new Baseclass.Object
     with null record;
   type F1_Class is access all F1'Class;
   type F1_Ptr is access all F1;

   procedure P1 (A : access F1) is abstract;

   function P2 (A : access F1) return String
     is abstract;

end Intf1;
```

and another interface:

```
with Baseclass;

package Intf2 is

   type F2 is abstract new Baseclass.Object
     with null record;
   type F2_Class is access all F2'Class;
   type F2_Ptr is access all F2;

   procedure P3 (A : access F2) is abstract;

   function P4 (A : access F2) return String
     is abstract;

end Intf2;
```

and we want to supply an object with both of these interfaces. We can create a new tagged type as follows:

```
with Baseclass;
```

```
with Intf1;
with Intf2;

package Impl1 is

   type Object is new Baseclass.Object with private;
   type Object_Ptr is access all Object;
   type Object_Class is access all Object'Class;
   subtype Object_Parent is Baseclass.Object;
   subtype Object_Parent_Ptr is Baseclass.Object_Ptr;

   function Cast (To_Cast  : access Object;
                  New_Type : in Ada.Tags.Tag)
                  return Object_Class;

   type Impl1_F1 is new Intf1.F1 with private;
   type Impl1_F1_Ptr is access all Impl1_F1;
   procedure P1 (A : access Impl1_F1);
   function P2 (A : access Impl1_F1) return String;

   type Impl1_F2 is new Intf2.F2 with private;
   type Impl1_F2_Ptr is access all Impl1_F2;
   procedure P3 (A : access Impl1_F2);
   function P4 (A : access Impl1_F2) return String;

private

   type Object is new Baseclass.Object with record
      My_F1 : Impl1_F1_Ptr;
      My_F2 : Impl1_F2_Ptr;
   end record;

   type Impl1_F1 is new Intf1.F1 with record
      My_Object : Object_Ptr;
   end record;

   type Impl1_F2 is new Intf2.F2 with record
      My_Object : Object_Ptr;
   end record;

end Impl1;
```

The above code creates an Object type. It also creates two other tagged types that define the interfaces required. The cast methods are defined as follows:

```
function Cast (To_Cast  : access Object;
               New_Type : in Ada.Tags.Tag)
               return Object_Class is
begin
   if (New_Type = Intf1.F1'Tag) then
      return Object_Class(To_Cast.My_F1);
   elsif New_Type = Intf2.F2'Tag) then
      return Object_Class(To_Cast.My_F2);
   else
      -- Call my parent's Cast method.
      return Cast(Object_Parent_Ptr(To_Cast),
                  New_Type);
   end if;
end Cast;
```

With this cast function, a variable of type Object can be cast to either of the interfaces using the following syntax:

```
This_F1 := Intf1.F1_Class(Cast(My_Object, F1'Tag));
This_F2 := Intf2.F2_Class(Cast(My_Object, F2'Tag));
```

Any methods defined for the new interfaces has access to the main object for the interfaces. Obviously, the creation routines (factories or whatever) should build the structure for the objects to point to each other.

Note that with this method (unlike Java), it is possible for interfaces to define code to run.

Chapter 2

Ada Structured Library Container Classes

This library contains the Ada Structured Generic Containers (ASGC) and the Ada Structured Object Containers (ASOC), which is a set of generic and object container classes that can be used from Ada95 programs. A number of general purpose containers are implemented. These are:

- Vector
- Doubly linked list
- Array List
- Tree
- Singly linked list
- Hash Table
- Heap
- Graph
- Directed Graph (DiGraph)
- Btree

Unlike many other types of containers, these are basic containers that can be used for a number of functions (stacks, queues, sets, bags, etc.). In the author's opinion, those

functions can be mapped to many types of containers, it is the use of the container that defines the function.

For most of the containers, several implementations are defined. The implementation types are:

- Fixed
- Expandable
- Dynamic

Each has it's own advantages and disadvantages, discussed later.

Each container defines two public tagged types, Object and Iterator. As well, each abstract container also defines these types. The Object is the actual container. The Iterator is the object used to iterate through the items in the container.

Two different package hierarchies are currently defined. One is the generic container class library, which defines a set of generic containers that can contain any type. Although the contained type may be any type (tagged or not), containers and iterators are tagged types.

The generic container class library is instantiated with the type in package Asoc_Collectible named Object_Class. The Asoc_Collectible package defines an object and a class pointer to that object. The containers are instantiated with the class pointer to the collectible object, so pointers to tagged types that inherit from Asoc_Collectible.Object can be contained using the object container class library. All the things contained in a container must be of the same exact type.

All tagged types defined by the ASGC derive from a single base class defined in the baseclass package.

2.1 Using Generic Containers

Generic containers exist in a generic package hierarchy. This means that the base package (Asgc) is a generic package and thus all it's children are generic packages. To instantiate a specific container, all the packages from the base package to the specific container must be instantiated.

So, for instance, if we want a sortable fixed vector and an ordered (not sortable) doubly-linked list both of floating point type, the following package could be used:

```
with Asgc.Ordered.Sortable.Vector.Fixed;
```

```
with Asgc.Ordered.Dlist.Dynamic;

package Float_Containers is

   package Float_Base is
     new Asgc(Contained_Type => Float);
   package Float_Ordered is
     new Float_Base.Ordered;
   package Float_Sortable is
     new Float_Ordered.Sortable;
   package Float_Sortable_Vector is
     new Float_Sortable.Vector;
   package FLoat_Ordered_Dlist is
     new Float_Ordered.DList;

   package My_Vector is
     new Float_Sortable_Vector.Fixed;
   package My_DList is
     new Float_Ordered_DList.Dynamic;

end Float_Containers;
```

Then we could write a procedure to use these containers:

```
with Float_Containers;
with Text_IO; use Text_IO;

procedure Use_Float_Containers is

   package Members renames Float_Containers.Float_Base;
   package Stack
     renames Float_Containers.Float_Ordered;

   My_Members : Members.Object_Class
     := new Float_Containers.My_Vector.Object
            (Size => 10);

   My_Stack : Stack.Object_Class
     := new Float_Containers.My_DList.Object;

   Val : Float;

begin
   Stack.Push(My_Stack.all, 1.0);
   Members.Add(My_Members.all,
```

```
                Stack.Get_At(My_Stack.all, 1));
     Stack.Push(My_Stack.all, 2.0);
     Members.Add(My_Members.all,
                Stack.Get_At(My_Stack.all, 1));
     Stack.Push(My_Stack.all, 3.0);
     Members.Add(My_Members.all,
                Stack.Get_At(My_Stack.all, 1));

     while (Stack.Member_Count(My_Stack.all) /= 0) loop
        Stack.Pop(My_Stack.all, Val);
        Put_Line("Popped " & Float'Image(Val));
     end loop;
  end Use_Float_Containers;
```

This method of instantiating all the packages may seem inconvenient, but it allows the operations on the container to be as generic as possible. For instance, if all you need is generic container operations, you should cast the type into and Asgc.Object_Class and use that particular type for the operations. This way, you can just rename the package you are using and you can use a different container with no other code changes.

It is also recommended that you rename the package to something you want to use the operation for. This also gives more flexibility by renaming the package. It also lets you name the package what you are using it for, such as the Stack in the example above.

2.2 Storage Management and Containers

All the containers can use storage management. Each generic container has a managed version, with "_Managed" appended to the end of the name. The managed version takes a storage management type and a storage manager variable as formal generic parameters, it will use that storage manager for all dynamically allocated data and for all access types it defines.

Several methods of specifying storage management were considered, and none of them seemed very good. The lesser of the evils was chosen. Some other options considered were:

Specify the storage manager at object creation - This would involve lots of nasty casting and would require a pointer to the storage manager (since they are limited types).

Specify the storage manager on every operation - This is similar to above except that the storage manager would have to be held for every allocated object so it could be properly deallocated.

Create a separate package with the storage manager as a required discriminant - This

requires pretty much complete code duplication since all the pointers must have the storage manager specified.

Pass in a dummy type as a generic parameter and use its storage manager - This makes things a little easier, but just seems too wierd to use.

The third approach was taken. Since the code is pre-processed by scripts anyway, it wasn't a big deal to add a managed version of each container.

However, I'd really like it to be an optional generic formal parameter, but there is no way to do this that I could figure out. If anyone has any ideas on how to do this, I'm listening...

2.3 Container Types

As mentioned before, the containers come in three different types:

Fixed - Fixed containers are defined with a static size (all currently implemented containers use an array). Fixed containers are guaranteed to not allocate memory dynamically (unless New_Iterator is called, which will dynamically allocate an iterator). Fixed containers generally take a size discriminant on creation, which is the number of element in the container.

Expandable - Expandable containers are much like fixed containers, except that they will expand their size when an item it added when they are full. They will not currently reduce their size (although that could feasible be added). These will only use dynamic allocation when they are initially created or when they expand. Expansion does require a copy, so it is not very efficient. Expandable containers generally take two discriminants: an initial size and an amount to increase the the container when it is expanded. The increase amount can be zero, then it will behave like a fixed container.

Dynamic - Dynamic containers use dynamic memory allocation for each item in the container. Dynamic containers do not generally take a size discriminant.

The container hierarchy is:

Asgc - The base generic container class. This hold functions that are available for all container classes and holds a few methods for operating on containers.

Asgc.Ordered - This package is the base of the package hierarchy for packages where each entry has an ordinal value that can be addressed. Ordered types can be used as vectors, lists queues, or stacks and have direct support for operations for those

functions. All the ordered containers have the same interface, the differency lies in the efficiency for performing certain operations.

Asgc.Ordered.Sortable - This package is the base of ordered containers whose contained items can be compared using >, <, etc. Thus they can be sorted. Otherwise, they are exactly like normal ordered containers. All ordered containers are also available as sortable containers.

Asgc.Ordered.Sortable.Vector, Asgc.Ordered.Vector - A variable sized array. Individual element are efficiently addressable, but insertion anyplace but the end is not efficient. Fixed and expandable containers are available.

Asgc.Ordered.Sortable.Alist, Asgc.Ordered.Alist - A variable sized array list. Individual element references are efficient (but not as efficient as a vector). Insertion is efficient of done at the beginning or at the end, but not in the middle. These make good stacks or queues if the maximum size is known or not very much expansion will be done. Fixed and expandable containers are available

Asgc.Ordered.Sortable.Dlist, Asgc.Ordered.Dlist - A doubly linked list. Insertion at any point is very efficient, but direct reference to an element number is not very efficient. Fixed, expandable, and dynamic containers are available. The fixed and exapandable containers use array indexes for the next and previous references, so copying them is efficient. They also do not do memory allocation for every new element. Referencing through array indexes is slower than direct pointers, though.

Asgc.Tree - A binary tree. It may be optionally balance (a balanced discriminant determines this). Fixed, expandable, and dynamic containers are available.

Asgc.List - A singly linked list. Insertion at any point is efficient and referencing the beginning and next values is efficient. Searching must be linear and iterators for this container cannot back up. Fixed, expandable, and dynamic containers are available. The fixed and exapandable containers use array indexes for the next and previous references, so copying them is efficient. They also do not do memory allocation for every new element. Referencing through array indexes is slower than direct pointers, though.

Asgc.Heap - A heap, ordered so the largest value comes out on top. As usual, insertion and removal or O(log n) operations. However, searching for a value in the heap is still O(n). Maybe someday another data structure will be laid over the heap (perhaps a balanced tree) to make searches quicker. Fixed, expandable, and dynamic heaps are available, much like trees. Note that the fixed and expandable structures take MUCH less store since their left, right, and up link values can be determined by their position in the array; link indexes are not needed. Note that the heap can be made to make the smallest value come out on top by reversing the ">", "<", ">=", and "<=" functions passed in to the generics.

Asgc.Hash - A hash table. A hash function must be provided, and the performance of the hash table depends greatly on the performance of the hash function. Note

that fixed, expandable, and dynamic hash tables do not work quite like other functions. A fixed hash table is a closed hash table with a given size. An expandable hash table is given a size and a maximum fill percentage. If the hash table will exceed the maximum fill percentage, it will instead create a new hash table (same size as the first) and chain them together, so both will be searched for a value. A dynamic hash table is an open hash table, the size of the table is specified. Note that hash tables can optionally hold more than one of the same value if the Allow_Duplicates discriminate is set to True. The Search and Search_Again functions will efficiently iterate through all the values in the table. A hash function must be supplied for hash tables. The quality of the hash function is very important, as well as the size of the hash table chosen. Read your data structures book on hash tables for more info.

Asgc.Graph - A graph container. This container can be used to implement arbitrary standard graphs (not directed graphs) or directed graphs. Internally the graph container uses a hash table for searching; it unfortunately has to expose some of this because it needs a hash table size and a hash function to do this, so those must be provided to the generics. The graph container is also a little unusual in that it actually has two container types: one for holding the graph nodes and one in each node to hold the links. Each of these may be individually chosen to be dynamic, fixed, or expandable, giving nine different containers that can be specified. Unfortunately, this makes things a little comples to specify. A separate section is dedicated to the graph container later and talks about how to use it. As with hash tables, the hash function and hash table size are important if the search functions are going to be efficient.

Asgc.Btree - A Btree container. This implements some variations of Btrees with B+ and B* extensions. Only a dynamic version is available. The other versions were just too hard to do. A Btree should be faster than a balanced binary tree for most operations, but does not expose its internal tree representation. It has a better worst-case performance than a hash table (all operations are logarithmic) but worse average performance than a hash table with a good hash algorithm.

All the containers are implemented as generic children in their hierarchy, so to instantiate a generic of a container, the whole hierarchy up to the required container must be instantiated. For instance, to create a dynamic tree if floats, the following would be used:

```
with Asgc.Tree.Dynamic;

package Float_Containers is
  new Asgc(Contained_Type => Float);
package Float_Tree is new Float_Containers.Tree;
package Float_Dyn_Tree is new Float_Tree.Dynamic;
```

Notice that the package specified to instantiate a child generic is the already instantiated parent generic, not the actual container name.

This same hierarchy is defined for object containers, except that the names are changed from Asgc to Asoc. The Asoc container's value is a class pointer to the Object type defined in Asoc_Collectible. So any type that inherits from Asoc_Collectible.Object can be put into an Asoc container. Note that if the values in the container are going to be compared (searched, sorted, put into a tree, hash table, etc.) they must be objects with the same tag. Otherwise, it would be difficult to compare them since it would raise an exception.

2.4 Container Operations

All containers support the following methods:

`Add(O, Val)` - Add the value "Val" to container "O". The add position may be any location, it depends upon the container.

`Delete(O, Val)` - Delete the value "Val" to container "O". If a container supports more than one entry of the same value, only one of the values is deleted.

`Member_Count(O) return Natural` - Return the number of items in the container.

`"="(O1, O2) return Boolean` - Compares two containers to see if they are equivalent. The meaning of this varies from container to container. In general, this isn't a very useful function for containers with nondeterministic positions (such as heaps or hash tables) but works fairly well for trees, lists, etc.

`Verify_Integrity(O)` - Performs internal checks on the integrity of the container's data structures. Raises an exception if the container has some internal problem.

`Copy(O) return Object_Class` - Make a copy of an object.

`Set_Callbacks(O, Cb)` - Set the callback object for the container. See the section on callbacks for more details.

`Generic_For_All(O)` - A generic method that will call the function specified in the generic for every member in the container. If the container is one with a specified order, the members will be processed in order.

`New_Iterator(O) return Iterator_Class` - Return a dynamically allocated iterator for the given object. Note that the value passed in must be a pointer to the object, therefore objects that are used with iterators must be able to be reference by access.

`New_Iterator(O) return Iterator` - Return an iterator for a given container. This is not a class-wide function, but one is provided for every container. Note that the value passed in must be a pointer to the object, therefore objects that are used with iterators must be able to be reference by access.

`Set_Container(Iter, O)` - Set the iterator's container. When an iterator is just declared, it does not intrinsically have a container associated with it. The container must be assigned either by assigning the iterator to another iterator or the return value of New_Iterator or by calling Set_Container.

2.4.1 Ordered and Sortable Container Object

Methods for all ordered and sortable containers follow. Note that when using one of the containers for a stack or a queue, the "At" operations work logically. Location 1 is the top of the stack, 2 is one under the top, etc. Location 1 is the head of the queue, etc.

`Add_At(O, Loc, Val)` - Add Val at the given location in the container. All members at that location and after in the container will be pushed forward one location.

`Set_At(O, Loc, Val)` - Set the value at the given location to the specified value. The Added callback is called for the new value then the Deleted value is called for the old value.

`Get_At(O, Loc) return Contained_Type` - Get the value at the specified location.

`Delete_At(O, Loc)` - Delete the specified location. All locations after that location are move one location back.

`Push(O, Val)` - Use the container like a stack and push one value onto the top of it.

`Pop(O, Val)` - Use the container like a stack and pop the value off the top of it.

`Enqueue(O, Val)` - Use the container like a queue and add an entry onto the head of it.

`Dequeue(O, Val)` - Use the container like a queue and take the tail entry off it.

Sorting

Generic sort packages are included for sortable containers. The following sort packages are available:

Asgc.Ordered.Sortable.Bubble_Sort - A standard bubble sort.

Asgc.Ordered.Sortable.Quicksort - A standard bubble sort.

Each of these provides two sort procedures:

`Sort(O)` - Sort the object. This routine will dynamically allocate iterators for its operations.

`Sort(O, Iter1, Iter2)` - Sort the object, but use the iterators provided instead of allocating them. The position of the iterators is destroyed.

2.4.2 Tree Container Object

No special methods are required for tree containers. Note that a tree can never hold more than one of the same value.

2.4.3 List Container Object

List containers add the following methods:

`Add_Head(O, Val)` - Add the given value to the head of the list.

`Add_Tail(O, Val)` - Add the given value to the Tail of the list.

2.4.4 Heap Container Object

Heap containers are used to find the Maximum (or minimum) value at the tree root, so special methods are defined to operate on the root:

`Get_Head(O) return Contained_Type` - Return, but do not remove the value at the root of the heap.

`Remove_Head(O, Val)` - Return in Val the value at the root of the heap then delete it from the heap.

2.4.5 Hash Table Container Object

Hash table containers add no special functions to the basic container operations.

2.4.6 Btree Container Object

Btree containers add no special functions to the basic container operations.

2.4.7 Graph Container Object

Graph containers provide the following additional methods:

`Set_Link_Callbacks(O, Cb)` - Set the callback object for the container. See the section on link callbacks for more details.

`Add_Link(O, From, To, Contents, Ignore_Dup := True)` - Add a link from the ”From” value to the ”To” value. If the link already exists and Ignore_Dup is False, an Item_Already_Exists exception is raised. If the link already exists and Ignore_Dup is True, the call does nothing. Otherwise, the link is added and the user contents of the link is set to the Contents parameter. Note that this will not change the exists contents if the link already exists and duplicates are not allowed and Ignore_Dup is set to True. Three routine are provided in this form, one with two iterators, one with the ”From” as an iterator and the ”To” as a value, and one with the ”From” as value and the ”To” as an iterator.

`Link_Exists(O, From, To) return Boolean` - Returns True if a link exists between the elements in the container that contain ”From” and ”To”.

2.5 Data Storage in Containers

The data used to maintain a container’s data structures is guaranteed to be unique to the container. If a container is copied by assignment or the Copy function, a complete new data structure is allocated to match the one from the source container. When a container is destroyed, all the data structures associated with it are freed automatically.

This does not apply to the values the user supplies the container. The user may use callbacks to provide a similar function.

Note that for set operations (unions, intersections, etc.) the containers assume that if a pair of values compare as equal, then they are replaceable, it doesn’t matter which of them is added, removed, etc. The user must be careful to honor this semantic if they expect set operations to work correctly.

2.6 Callbacks

The user may define a callbacks type (inheriting from the Callbacks type in Asgc) that has methods that will be called when operations are performed on the values in the container. The following methods are provided:

`Added(Cb, O, Val)` - An item was added to the container with an explicit add operation of some type (Add, Enqueue, etc).

`Copied(Cb, O, Val)` - An item was copied from one container to another. this is only called for explicit copies between containers, not for copy operations inside the container.

`Deleted(Cb, O, Val)` - An item was deleted from the container.

The containers are careful to perform the Added method before they perform the Deleted method if values are being replaced, so it is safe for the user to keep a reference count in the object to tell how often it is being used. Also, the user may replace the object with a new object, but must be VERY careful to keep the ”value” of the object (when compared) exactly the same as the object being replaced.

These callbacks are primarily intended for automatic storage reclamation. So, for instance, the value may be a pointer to some data structure that the user dynamically allocates and puts in the container. The user can add a reference count initialized to zero in the data structure. On every Added or Copied call, the user should increment the reference count. On every Deleted call, the user should decrement the reference count and if it reaches zero the value can be freed. This removes the difficult operation of freeing memory from the main body of the user's code (often called “garbage colletion lite”.

Another way to do this is to have Added do nothing, Copied make a copy of the value, and Deleted always free the value. Since the value it is passed ”in out”, the user can modify the value passed in and the value returned will be put into the new container.

2.7 Iterators

Iterators are objects that hold some position within a container and can perform operations based upon that position. This implies, of course, that containers have some order, but the order is arbitrary and not always reversable.

Because of the arbitrariness of the order, an operation on a container that modifies the container's contents will invalidate all iterators that reference the container, if they are used without their position being reset an exception will be raised. If the operation that

modifies the container is performed using an iterator, that specific iterator will still be valid and will reference the ”Next” value in the container. The only exception is when the last item in the container is deleted, the iterator cannot back up and will thus be invalid.

Iterator operations that move the position of the iterator will generally return an End_Marker to tell if the iterator attempted to move past either end of the container.

When an iterator is created, a class pointer to the base object defined in Asgc is required. This means that all containers that need to be iterated must be dynamically allocated or aliased. It would be nice if this was not the case, but it is very difficult to reference one object from another without a pointer.

Basic methods on iterator are:

`Add(Iter, Val)` - Adds the item to the container the iterator references and positions the iterator on the added item.

`First(Iter, Is_End)` - Set the iterator position to the first item in the container. If the container is empty, the Past_End will be returned in Is_End and the iterator will not be valid.

`Next(Iter, Is_End)` - Move the iterator to the next item in the container. If the current item is the last item in the container, then the iterator will not be modified (it will still be positioned on the last item) and Is_End will be set to Past_End.

`Delete(Iter, Is_End)` - Remove the value reference by the iterator from the container. If the value is the last item in the container, Is_End will be set to Past_End and the iterator will be invalidated. Otherwise, the iterator will be positioned on the item after the deleted item and Is_End will be set to Not_Past_End.

`Is_Same(Iter1, Iter2) return Boolean` - Returns True if the iterators are positioned on the same location in the same container.

`Get(Iter) return Contained_Type` - Return the value at the current position of the iterator.

`Get_Incr(Iter, Val, Is_End)` - Return the value at the position of the iterator then move the iterator to the next item in the container. The standard Next() method semantics apply to this function, too.

`"="(Iter1, Iter2),"="(Iter1, Val),"="(Val, Iter1)` - Normal comparison operations on iterators do not compare the iterators themselves but the values they reference.

`Search(Iter, Val, Found)` - Search for the specified value from the beginning of the container. If the value is found, then Found is set to True and the iterator is positioned on the position holding that value. Otherwise, Found is set

to False and the iterator will be invalid. The efficiency of this operation depends on the specified container.

`Search_Again(Iter, Found)` - Search again for the value at the current position of the iterator. Calling Search() and then Search_Again() until False is returned in Found is guaranteed to get all items in the container with the specified value.

`Entry_Count(O, Val) return Natural` - Return the number of times the specified value is in the container.

`Union(Dest, O1, O2)` - Perform a set union of O1 and O2 and return the result in Dest. These values are class pointers to object, not straight objects, because iterators must be used to work on the containers. This routine works even if any of the containers are the same container. Note that this is a pure set union, if either of the source containers contain more than one of the same value then the result of this will be invalid or an exception might be raised.

`Intersection(Dest, O1, O2)` - Perform a set intersection of O1 and O2 and return the result in Dest. These values are class pointers to object, not straight objects, because iterators must be used to work on the containers. This routine works even if any of the containers are the same container. Note that this is a pure set intersection, if either of the source containers contain more than one of the same value then the result of this will be invalid or an exception might be raised.

`Bag_Union(Dest, O1, O2)` - Perform a bag union of O1 and O2 and return the result in Dest. These values are class pointers to object, not straight objects, because iterators must be used to work on the containers. This routine works even if any of the containers are the same container. Since a bag can contain multiple of the same value, the destination must be able to hold duplicate values. Otherwise, an exception will probably be raised.

`Bag_Intersection(Dest, O1, O2)` - Perform a bag intersection of O1 and O2 and return the result in Dest. These values are class pointers to object, not straight objects, because iterators must be used to work on the containers. This routine works even if any of the containers are the same container. Since a bag can contain multiple of the same value, the destination must be able to hold duplicate values. Otherwise, an exception will probably be raised.

2.7.1 Ordered and Sortable Container Iterator

Iterators for ordered and sortable containers have the following methods:

`Last(Iter, Is_End)` - Set the iterator position to the last position in the container. Is_End will be set to Past_End if the container is empty.

`Prev(Iter, Is_End)` - Set the iterator position to the previous position in the container. If the iterator is at the first item in the container, the iterator is not modified and Is_End is set to Past_End.

`Set_Loc(Iter, Loc)` - Set the iterator position to a direct location. If the location is not valid (or the container is empty) a Constraint_Error will be raised.

`Get_Loc(Iter) return Natural` - Return the current direct location of the iterator.

`Is_After(Iter1, Iter2) return Boolean` - Return True if the position of Iter1 is after the position of Iter2 in the container. The iterators must reference the same container.

`Is_Before(Iter1, Iter2) return Boolean` - Return True if the position of Iter1 is before the position of Iter2 in the container. The iterators must reference the same container.

`"-"(Iter1, Iter2) return Integer` - Return the difference between the location of Iter1 and the location of Iter2.

`"+"(Iter1, Offset) return Iterator` - Return a new iterator at the given offset from the position of Iter1. If the position goes beyond the range of the container, a Constraint_Error will be raised.

`"-"(Iter1, Offset) return Iterator` - Return a new iterator at the given offset from the position of Iter1. If the position goes beyond the range of the container, a Constraint_Error will be raised.

`Swap(Iter1, Iter2)` - Swap the values at the given locations in the iterator. Note that no callback is called for this if the operation occurs in the same container. If Iter1 and Iter2 reference different containers, Added callbacks are called for both values on their destination containers and then Deleted callbacks are called for the source containers.

`Add_After(Iter, Val)` - Add the given value after the iterator's current position. After this operation the iterator will be positioned on the newly added value.

`Add_Before(Iter, Val)` - Add the given value before the iterator's current position. After this operation the iterator will be positioned on the newly added value.

`Set(Iter, Val)` - Set the value of the iterator's position. The Added callback is called for the new value then the Deleted value is called for the old value.

`Get_Decr(Iter, Val, Is_End)` - Return the value at the position of the iterator then move the iterator to the previous item in the container. The standard Next() method semantics apply to this function, too.

Iterators for sortable containers can be compared ordinally, so they add the following methods. These comparison methods (as with the `"="` operator) compare the values the iterators are positioned at, not the position of the iterator.

```
">"(Iter1, Iter2)
">"(Iter, Val)
">"(Val, Iter)
"<"(Iter1, Iter2)
"<"(Iter, Val)
"<"(Val, Iter)
">="(Iter1, Iter2)
">="(Iter, Val)
">="(Val, Iter)
"<="(Iter1, Iter2)
"<="(Iter, Val)
"<="(Val, Iter)
```

2.7.2 Tree Container Iterator

Iterators for trees can be used to traverse the tree as a tree. Note that the standard iterator methods First and Next will do an in-order traversal of the tree.

`Root(Iter, Is_End)` - Move to the root node of the tree. If the tree is empty, Is_End will be set to Past_End.

`Left(Iter, Is_End)` - Move the iterator to the left subtree of the iterators current position. If the left subtree does not exist, Is_End will be set to Past_End and the iterator will remain in the same position.

`Right(Iter, Is_End)` - Move the iterator to the right subtree of the iterators current position. If the right subtree does not exist, Is_End will be set to Past_End and the iterator will remain in the same position.

`Up(Iter, Is_End)` - Move the the parent of the current node. If the current node is the root node, then Is_End will be set to Past_End and the position of the iterator will be unchanged.

`Last(Iter, Is_End)` - It's as easy to do a reverse in-order traversal as a forward in-order traversal, so it is possible to go backwards through the tree as well as forwards. This moves to the largest entry in the tree. If the tree is empty, Is_End will be set to Past_End;

`Prev(Iter, Is_End)` - Set the iterator position to the previous position in the Tree. If the iterator is at the first item in the tree, the iterator is not modified and Is_End is set to Past_End.

Get_Decr(Iter, Val, Is_End) - Return the value at the position of the iterator then move the iterator to the previous item in the tree. The standard Prev() method semantics apply to this function, too.

Iterators for tree containers can be compared ordinally, so they have the following methods. These comparison methods (as with the "=" operator) compare the values the iterators are positioned at, not the position of the iterator.

```
">"(Iter1, Iter2)
">"(Iter, Val)
">"(Val, Iter)
"<"(Iter1, Iter2)
"<"(Iter, Val)
"<"(Val, Iter)
">="(Iter1, Iter2)
">="(Iter, Val)
">="(Val, Iter)
"<="(Iter1, Iter2)
"<="(Iter, Val)
"<="(Val, Iter)
```

2.7.3 List Container Iterator

Iterators for lists add the following methods to the basic iterator methods:

Add_After(Iter, Val) - Add the given value after the iterator's current position. After this operation the iterator will be positioned on the newly added value.

Add_Before(Iter, Val) - Add the given value before the iterator's current position. After this operation the iterator will be positioned on the newly added value.

Set(Iter, Val) - Set the value of the iterator's position. The Added callback is called for the new value then the Deleted value is called for the old value.

2.7.4 Heap Container Iterator

Iterators for heaps add no methods to the basic iterator methods.

2.7.5 Hash Table Container Iterator

Iterators for hash tables add no methods to the basic iterator methods.

2.7.6 Btree Container Iterator

Iterators for Btress add the following methods to the basic iterator methods:

`Prev(Iter, Is_End)` - Set the iterator position to the previous position in the Btree. If the iterator is at the first item in the tree, the iterator is not modified and Is_End is set to Past_End.

`Get_Decr(Iter, Val, Is_End)` - Return the value at the position of the iterator then move the iterator to the previous item in the tree. The standard Prev() method semantics apply to this function, too.

As well as the following ordinal comparisons are defined:

```
">"(Iter1, Iter2)
">"(Iter, Val)
">"(Val, Iter)
"<"(Iter1, Iter2)
"<"(Iter, Val)
"<"(Val, Iter)
">="(Iter1, Iter2)
">="(Iter, Val)
">="(Val, Iter)
"<="(Iter1, Iter2)
"<="(Iter, Val)
"<="(Val, Iter)
```

2.7.7 Graph Container Iterator

Iterators for graphs can be used like a normal iterator, but have within them another iterator to iterate over the links a node has. So each node in a graph has a set of links that the user may iterate over with the following special operations. Note that changing the links in a node does not invalidate other iterators in the container. Also note that if the iterator's node reference is changed, the link iterator of the iterator is invalidated.

`Add_Link(From, To, Contents, Ignore_Dup := True)` - Add a link from the "From" iterator to the "To" Iterator. If the link already exists and Ignore_Dup is False, an Item_Already_Exists exception is raised. If the link already

exists and Ignore_Dup is True, the call does nothing. Otherwise, the link is added and the user contents of the link is set to the Contents parameter. Note that this will not change the exists contents if the link already exists and duplicates are not allowed and Ignore_Dup is set to True.

`Delete_Link(Iter, Is_End)` - Delete the link the iterator currently references. If the last link in the container is deleted, Is_End will be set to Past_End and the link part of the iterator will be set to invalid (the main part of the iterator is still valid, though).

`Find_Link(From, To) return Iterator` - Find a link from the "From" iterator's node to the "To" iterators node and return an iterator that references the link from the "From" iterator. So the returned iterator will reference the same node as the "From" iterator but the link reference of the iterator will reference the link that goes to the "To" iterator. The "From" and "To" iterators must reference the same container. If the link is not found, an Item_Not_Found exception will be raised.

`Find_Link(From, To, Found)` - Find a link from the "From" iterator's node to the "To" iterators node and move the "From" iterator so that its link reference references the link to the "To" iterator's node. If the link is not found, the From iterator will not e changed and the Found parameter will be set to False. If the link is found, Found will be set to True. Several versions of this exists with various combinations of iterator and contained types.

`Find_Link_Again(From, Found)` - Search for the next link that has the same destination as the current link. This is only useful for graphs that support duplicate links, with this function all the links to a specific destination can be found. If a link to the same destination is not found, Found will be set to False and From will not be modified. Otherwise, From will reference the next link.

`Link_Exists(From, To) return Boolean` - Returns True if a link exists from From to To, False if it doesn't exist. Several versions of this exists with various combinations of iterator and contained types.

`First_Link(Iter, Is_End)` - Move the link reference for the iterator to the first link in the node. If the node has no links, Is_End will be set to Past_End and Iter will be unchanged.

`Next_Link(Iter, Is_End)` - Move the link reference for the iterator to the next link in the node. If the iterator references the last link in the node, Is_End will be set to Past_End and Iter will be unchanged.

`Follow_Link(Iter) return Iterator` - Follow the current link of "Iter" and return and iterator that reference the node the link is to.

`Follow_Link(Iter)` - Set Iter to the node that its link references when it is passed in.

`Link_Count(Iter)` - Return the number of links in the node the iterator references.

`Get_Link(Iter)` - Return the user data value for the link the iterator references.

`Set_Link(Iter, Val)` - Set the user data value for the link the iterator references.

2.8 Using Graph Containers

As mentioned before, graph containers are more difficult to create because of the multitude of options available. In this example, I will be using the dynamic graph container, the fixed or expandable containers may be easily substituted.

A generic graph container takes two formal parameters to instantiate it, a link container and an iterator for the link container. These two parameters must be compatible (you can't use a dynamic link container with a fixed link iterator, for instance). The link container and iterator passed in determine if the graph nodes are dynamic, expandable, or fixed.

Dynamic graph nodes must be instantiated with a child package named Asgc.Graph.Dynamic.Links.Dynamic, the names, it contains the types Graph_Link and Graph_Link_It. So, to instantiate a dynamic graph container with dynamic nodes, use something like:

```
package Base is new Asgc(Contained_Type => Integer);
package Graph_Base is new Base.Graph
  (Do_Hash            => Hash_Integer,
   Link_Contained_Type => Boolean);
package Dyn_Graph_Base is new Graph_Base.Dynamic;
package My_Graph_Links is
  new Dyn_Graph_Base.Links.Dynamic;
package My_Graph is new Dyn_Graph_Base.Graph
  (My_Graph_Links.Graph_Link,
   My_Graph_Links.Graph_Link_It,
   Allow_Duplicate_Links => False);
```

And My_Graph will be a package that the graphs can be used from.

Fixed and expandable graph node are more complicated because they have discriminants that have to be handled. The user cannot just subtype some type and constrain a type with a size because that cannot be passed in as one of the package discriminators for the graph package; Ada95 does not allow that. Instead, the user must instantiate a generic package with the required sizes. For instance, in the above example, replace the last two lines with:

```
package My_Graph_Links is
  new Dyn_Graph_Base.Links.Fixed(Size => 20);
package My_Graph is new Dyn_Graph_Base.Graph
  (My_Graph_Links.Graph_Link,
   My_Graph_Links.Graph_Link_It,
   Allow_Duplicate_Links => False);
```

or

```
package My_Graph_Links is
  new Dyn_Graph_Base.Links.Expandable
  (Initial_Size => 20, Increment => 30);
package My_Graph is new Dyn_Graph_Base.Graph
  (My_Graph_Links.Graph_Link,
   My_Graph_Links.Graph_Link_It,
   Allow_Duplicate_Links => False);
```

For fixed or expandable nodes.

Digraphs are specified by using DiGraph instead of Graph in the last generic package instantiation. For instance, taking the first example above, replacing the last package with the following will make a digraph:

```
package My_DiGraph is new Dyn_Graph_Base.DiGraph
  (Dynamic_Graph_Link, Dynamic_Graph_Link_It);
```

Note that, obviously, if you follow a link in a digraph you don't necessarily have a link back. Also, since you don't have a link back, links in to a digraph to not eat up the normal links. However, there is a hidden link back to all the links to a node and it is the same size as the normal links, so a fixed node has a limit on the number of other nodes that can link to it.

Note that links themselves support having data items on them (like the Link_Contained_Type above). They are added with Add_Link and can be fetched with Get_Link or modified with Set_Link.

Also note that duplicate links between nodes can be optionally supported by setting Allow_Duplicate_Links to True as shown above.

2.8.1 Creating Your Own Graph Link Nodes

You can create your own graph link node types if you need some special properties (for instance, you have a lot of links in each node and it needs to be more efficient

than a linked list). However, you must be VERY careful to follow the semantics of the interface carefully. See the package Asgc.Graph.Links for details on doing this.

2.8.2 Link Callbacks for Graph Containers

For graph containers, the links themselves carry data values and these may have callbacks just like regular values in the graph. See the Callbacks section for more details. Since link values are not used by the container classes, the values on links may be changed arbitrarily, unlike the main container callbacks.

IMPORTANT – With regular graphs, these functions will be called TWICE for every link added, deleted, or copied, since the link goes both directions. The methods must account for this properly, especially if they generate a new copy, since each link direction will have a different copy in that case. The methods should not assume any calling order, so doing something like ”Odd adds are real and even one just use the last value” is dangerous. This does NOT apply to directed graphs, they will only call the functions once per link since the links only go one way.

Chapter 3

General ASL Library Routines

Some general library routines are defined by the ASL library. These are routines that are often used in Ada95 code.

3.1 Semaphores

Ada95 does not define semaphore directly, but they are easily defined using protected types. Please don't use semaphores unless you really need them. Protected types are much safer and easier to use. However, if you need to block during an operation, need a wakeup function that cannot be done with a rendevous, need a counting semaphore, or need a nested semaphore, these packages are for you.

The package Asl.Semaphore contains an abstract semaphore type that all semaphores derive from and implement. It contains the whole interface to semaphores, with the following methods:

`Take(S)` - Claim the semphore S. This will block until S becomes free and then claims S.

`Give(S)` - Release a semaphore so it can be taken again.

`Try_To_Take(S, Success, Timeout)` - Attempt to take the semaphore, but fail the operation if it takes longer than Timeout. If the operation is successful, Success is set to True, Otherwise it is False. Timeout defaults to 0.0.

Four semaphore types are declared in the following packages:

Asl.Semaphore.Binary - A standard binary semaphore.

Asl.Semaphore.Counting - A standard counting semaphore. The initial value is specified when the variable is declared.

Asl.Semaphore.Nested - A binary semaphore that allows the same task to take it multiple times. The number of claims is tracked and the semaphore must be given the same number of times. A Tasking_Error will be raised if a task other than the owner tries to give the semaphore.

Asl.Semaphore.Nested_Prio - Like a nested semaphore, but it implements a priority inheritance algorithm. DANGER - These semaphores don't work right unless setting the dynamic priority of a task sets it immediately. GNAT doesn't do this by under Linux. I'm thinking of a way to fix it, but nothing has come to me yet.

3.2 Leak Detection Storage Pool

Asl.Leak_Detect_Pool defines a storage manager that can be used to detect leaks in your code. If you specify this storage manager, it will keep a linked list of all data items that have been allocated but not freed using it. After you are done, you can use the storage manager to iterate over all the elements that have not been freed. This means, of course, that you must free all your data :-).

The iterator returns an address and a size. If your system is deterministic about its memory allocation, you can set a conditional breakpoint in the Allocate routine of this package right before it sets the return address that has the leaked address from a previous run. Then you can move up stack levels to see where the memory was allocated. The same address can be returned again if it is freed, so be aware that you must catch the last allocation.