Implementing the C++ Standard Template Library in Ada 95

Úlfar Erlingsson Alexander V. Konstantinou
Computer Science Department*
Rensselaer Polytechnic Institute
Troy, NY 12180

January 31, 1996

Abstract

The Standard Template Library (STL), a recent addition to the ANSI C++ standard, "provides a set of well structured generic C++ components that work together in a seamless way" [SL94]. The popularity of STL stems from its combination of an orthogonal design, solid theoretical foundation, and strong emphasis on efficiency. This paper presents a design scheme for implementing the C++ STL library components in Ada, using features introduced in the 1995 Ada standard [Ada95a]. Discussion is based on a prototype Ada 95 implementation, segments of which illustrate the paper. This work was prepared under the supervision of Dr. David Musser at Rensselaer Polytechnic Institute.

*Email: {`ulfar,akonstan`}`@cs.rpi.edu`

Contents

1 Introduction

The C++ Standard Template Library (STL), a recent addition to the ANSI language standard [LRM], "provides a set of well structured generic C++ components that work together in a seamless way" [SL94]. The design of the library emphasizes component decomposition, orthogonality, and strict semantic and complexity requirements. Efficiency is another important consideration, and the library components closely exploit the C++ template mechanism to minimize run time overhead. The library approach results in a design based on a solid theoretical foundation, while remaining natural and easy to grasp. This paper will present an implementation scheme for porting the the C++ Standard Template Library to Ada 95.

The availability of generics in Ada, as early as 1983, has long attracted researchers of generic library components, including the designers of the C++ Standard Template Library [MS89]. Language restrictions in the 1983 standard, however, prevented the implementation of some of the main STL components. Now that the long awaited revised 9X standard for Ada has been finalized into Ada 95, many of the earlier restrictions have been lifted, making it appropriate to re-examine the issue.

The following sections will briefly introduce the main components of STL, highlight the relevant changes in the Ada 95 standard, and propose a design for an Ada STL implementation. For demonstrational purposes, a representative subset of the STL components has been implemented in Ada 95. The aim has been to show the feasibility of translating the component interrelations from the C++ model to the Ada model. In this sense, little attention had been paid to important details such as run-time exceptions and efficient memory allocation. Details concerning this prototype implementation can be found in the appendix.

2 C++ STL Overview

The C++ Standard Template Library, see [SL94], provides a set of generic components and a consistent interface between them. The main considerations in the design of STL were that all algorithms and data structures work in a similar manner to their native C++ equivalents, that the implementations be efficient, and that the library should provide an easy-to-grasp orthogonal interface based on a sound theoretical foundation.

2.1 STL Categories

STL groups its components into a few distinct categories. Following is a brief listing and discussion of each category. Interested readers should refer to [SL94] for a more detailed discussion.

Containers Containers in STL are generic in their element types; associative containers are also generic in an ordered key type. STL provides vectors, lists and deques as sequence containers, and sets, multisets, maps and multimaps as sorted associative containers.

Iterators Iterators in STL are based on the C++ concept of accessing elements in data structures through pointers. STL iterators provide the functionality of pointers for STL containers. Informally, iterators are classified into the following categories based on which operations they support:

Input/Output There is no decrement. Input iterators are read-only. Output iterators are write-only. You should not assign twice to the same output iterator position. Both iterators categories only support single pass algorithms.

Forward Supports increment, and read/write with no restrictions.

Bidirectional Supports increment/decrement, and read/write with no restrictions.

Random Access Supports all operations a C++ pointer would, i.e. increment/decrement, read/write, pointer-arithmetic, comparisons, and overloaded index operator (`operator[]`).

Another iterator concepts supported in the C++ STL are constant and reverse iterators. Constant iterators allow for the provision of read-only access to container elements. Reverse iterators have the functionality suggested by their name; e.g., the incrementing and decrementing operations have their meanings interchanged.

Function Objects Function objects offer an encapsulation of a function, and perhaps some state information, into an object. Function objects in C++ STL are used to support the instantiation of generic components with function arguments. However, since function objects can also encapsulate state information, they allow for a form of functional programming in C++ STL.

Algorithms The STL generic algorithms provide various common algorithms on iterators and function objects. Examples include `for_each`, which calls a function object with each element in a range, `reverse`, which reverses the contents of a range by repeated swaps, and the more complicated `sort`, which sorts a range given a comparator function object.

Adaptors Adaptors compose new components from existing ones. There are tree main types of adaptors:

- **Containers** The STL sequence containers are adapted into stacks, queues and priority_queues.
- **Iterators** The most important STL iterator adaptors are perhaps the ones which provide reverse iterators based on normal ones, and those which provide insert iterators for containers.
- **Function Objects** Function objects can be adapted in several useful ways in the style of functional programming. Function object parameters can be bound, composing new functions objects taking fewer parameters. Function objects can also be negated. There are also adaptors allowing the use of pointers to functions.

2.2 C++ Considerations

Even though it is written in C++, an object-oriented programming language, STL does not use inheritance to any great extent. Instead it relies heavily on composition of its generic components to provide for specialization. Thus stack does not inherit from list, deque or vector. It is instead a generic composition from any container providing the required functionality.

The components of STL make heavy use of the C++ template facility for generic programming. In C++, when defining a class or function using templates, you implicitly require any operations or functions used with arguments of the generic type to be defined for the instantiation type. This is not checked until the template code is instantiated with an actual type.

This approach to generic programming in C++ is very flexible at the cost of not being safe. It is not always clear when instantiating a template which operations or functions are going to be used in the instantiated code. This "late compile-time binding" of templates can easily lead to various unexpected behavior. This problem is highly aggravated by the implicit type-casting often performed in C++.

3 STL in Ada 95

The 1995 Ada standard adds numerous new features to the 1983 standard, some of which make an Ada implementation of the Standard Template Library possible. One of the most important extensions, for our purposes, involves access types. Unlike Ada 83, where access type variables can only refer to objects created dynamically, Ada 95 has a general access type. Instances of types defined as `aliased` can now be referenced through the `Access` attribute. Remaining restrictions on pointer arithmetic, which can lead to inefficiencies in the implementation of random access iterators, can be circumvented through the C language interface library package. These issues are further discussed in Section 6.2 on vector containers.

A second change that simplifies presentation and adds to safety is the ability to declare generic formal packages. Thus, a generic package can be instantiated with one of the parameters being an instance of a certain generic package. This feature allows for the definition of signature packages that help to cleanly define and enforce the interface between the library components. Moreover, signature packages can add to safety by requiring explicit instantiation of their generic formal component operators, instead of the implicit requirement by use. (as is the case in C++). The next section, on iterators, expands on this feature.

There are numerous other changes, notably those involving object oriented programming features, which are not used by this Ada STL implementation. Interested readers are referred to [Ada95a] and [Ada95b].

4 Iterators

In C++ STL, container iterators are objects providing a certain functionality imported from their container. Depending on this functionality five iterator categories are defined: input, output, forward, bidirectional, and random access iterators. These categories form a hierarchy, with forward iterators satisfying all the input and output iterator requirements, bidirectional iterators satisfying all those of forward iterators, and random access iterators satisfying all those of bidirectional iterators. At compile time, iterator objects are mapped into simple pointers, and their operations inlined, thus providing run-time efficiency. Because iterators provide the only interface between containers and algorithms, it is essential that they be compiled efficiently without the need for any run-time binding.

The above scheme cannot be directly ported to Ada because types used to instantiate a generic package do not pass along the operations defined on them. Although it is possible for the generic package to require that the operations be passed along explicitly by the caller, this enforces a copious style. For implementing STL in Ada, a better solution is to use an intermediate generic package to define a signature. This package is subsequently used as a generic formal by an algorithms package. For every iterator category, a generic signature package is defined and acts as the mapping abstraction. The resulting hierarchy of algorithms packages with iterator formal parameters reflects that of iterators, for example, random access algorithms, bidirectional algorithms, and input-output algorithms.

A signature package `forward_iterators` is shown here:

```
generic type f_value_type is private;
        type f_iterator is private;
  with procedure Assign(i: in f_iterator; v: f_value_type);
  with function Val(i: in f_iterator) return f_value_type;
  with procedure Inc(i: in out f_iterator);
package forward_iterators is end;
```

Any algorithm using forward iterators has to restrict itself to the three operations defined above. Respectively, any container that exports iterators to be used with forward algorithms has to implement all three operations, something which is checked by the Ada 95 compiler. There is no way to obtain the corresponding checking with C++ templates.

Our current implementation does not provide constant or reverse interators. The intention is to provide constant iterators through another set of signature functions which exclude the `Assign` operation. Reverse iterators, however, are to be provided though an iterator adaptor.

5 Function Objects

Since packages can have function or procedure parameters, function objects play a much less significant role in an Ada implementation of STL than in the C++ one. There is in fact, in Ada, no need to provide any of the predefined function objects in STL. However, the concept of associating state with a function, or procedure, is still a very useful one.

5.1 Packages as Function Objects

Ada allows a package to contain state information accessible only to the functions, or procedures, in the package. By using this we can create packages having side-effects, similar to function objects in C++ STL and to closures in functional languages. However by using only this approach we are constrained to having only one instance of any function object in our code. Therefore we cannot, for example, have two accumulator objects with two different sums.

We can, however, easily get away from this limitation by making the package generic. This allows us to construct as many instances of the package as we require. An example of the specification of a simple accumulator package follows:

```
with unary_procedures;
generic type T is private;
        Nil: T;
        with function "+"(x: T; y: T) return T;
package accumulator is
  function Sum return t;
  procedure Add(x: in T);
  procedure Reset;
  package signature is new unary_procedure(T, Add);
  private sum_value: t := Nil;
end accumulator;
```

5.2 Function Signatures

As is the case with iterators, signature packages are used to provide a standard interface to functions and procedures. The Ada implementation has to provide such generic packages for both functions and procedures, something which C++ STL does not require, since a C++ procedure is a function with a `void` return type. This somewhat awkward code duplication is required in several parts of the Ada STL library, because of the Ada distinction between functions and procedures.

The Ada implementation has the following signature packages: `unary_procedures`, `binary_procedures`, `unary_functions` and `binary_functions`. In C++, mode information is part of the instantiation type for a generic function, whereas in Ada it is a syntactic construct. This seems, at first glance, to force us to provide different signature packages for procedures of all possible modes. However, since procedures are not used often in STL, except in read-only situations, we only provide `in`-mode procedures in this implementation.

Following is the signature package for unary functions:

```
generic type argument_type is private;
        type result_type is private;
```

```
        with function Op(x: argument_type) return result_type;
    package unary_functions is end;
```

6 Containers

For this study we implemented two of the STL containers, lists and vectors. This allowed us to make use of the full range of iterators, and to create examples for all types of algorithms.

Perhaps the biggest difference between C++ STL and Ada STL containers stems from the fact that Ada packages do not define object types in the sense that C++ classes do, but rather represent the traditional concept of abstract data types. Thus when using a list, one passes an instance of a special `List` type to functions defined in the `Lists` package. This means the C++ "dot"-notation `x.y()`, where `x` is an instance of a class with a member function `y()`, has a corresponding `y(x)` Ada equivalant.

The container components instantiate all possible iterator signature packages as subpackages, e.g., `lists` has `input_iterator`, `output_iterator`, `forward_iterator`, and `bidirectional_iterator`, all as subpackages. This relieves the user from having to instantiate any iterator signatures, since they are predefined in each container package. The user need only pass the correct subpackage to an algorithms package to instantiate the algorithms for a container.

6.1 Lists

The Ada implementation of lists is modeled very closely on the C++ reference implementation in [STL]. Lists are doubly linked and circular with a sentinel node before the beginning and after the end.

In order to keep the Ada implementation as simple as possible, only a subset of the member functions of lists were implemented. These functions are `empty`, `size`, `push_back` and `pop_back`. There is also a constructor function `Create` in the Ada version.

The list iterator is implemented in the subpackage `Lists.Iterators`. An iterator is defined there as a normal Ada `access all` type to an `aliased` cell in an instance of the `List` type. While not sufficient for vectors, this suffices for lists since lists do not need to support pointer arithmetic. The C++ STL functions `begin()` and `end()` are implemented in `Iterators` as `Start` and `Finish`. The list iterator satisfies the signature of bidirectional iterators, with the functions `Inc`, `Dec`, `Val` and `Assign` allowing increment/decrement, dereferencing and assignment on iterators.

Following is an overview of the Ada specification for lists:

```
with input_iterators, ..., bidirectional_iterators;
generic type T is private; ...
package Lists is
  subtype Value_Type is T;
  type List is private;
  ⋮
  function Create return List;
  ⋮
  package Iterators is
    type Iterator is private;
    procedure Assign(i: in Iterator; v: in Value_Type);
    function Start(l: List) return Iterator;
    ⋮
    private type Iterator is record ... end record;
  end Iterators;
  package input_iterator is new input_iterators
    (Value_Type, Iterators.Iterator, Iterators.Inc, Iterators.Val);
  ⋮
  private type List is record ... end record; ...
end Lists;
```

6.2 Vectors

An STL vector supports random access iterators, and is a generalization of a dynamic array. Besides the usual array operations, there is support for linear time `insert`, nd `erase` in the middle, as well as amortized constant time `insert` and `erase` at the end of the vector. Storage management is not visible to the user, although tuning parameters may be supplied.

The main issue in an Ada implementation of the vector container involves the choice of the iterator type. Even with `aliased` types, Ada does not allow for pointer arithmetic. Therefore, one cannot increment a pointer to a vector element, or subtract two pointers. One solution would be to define the vector iterator as a record containing an index position and a pointer to the array. Getting the value (dereferencing) of the iterator would involve the computation of the array address for the given index. A second solution employs the C interfaces library package for unchecked conversions between array pointer types and address types. The second solution provides the most direct mapping to the C++ STL vector container, and is the one used in our Ada implementation.

It is important to note that the choice in iterator implementation does not affect the other components of the Ada STL implementation. One can argue that basic containers supplied with an Ada STL implementation should use the more efficient, if less secure, solution. Additional container construction can use an adaptor approach, wrapping the efficient implementation with safety checks. This discussion pertains to the internal validity of the container implementation. There is a separate issue concerning the extent of run-time range error checking, discussed earlier, which applies equally to both solutions.

The Ada specification for the vector container is similar to the one for lists, differing in the iterator specification:

```
generic type T is private;
package vectors is
  subtype Value_Type is T;
  ⋮
  package Iterators is
    type Iterator is private;
    ⋮
    function "-"(i: in Iterator; j: in Iterator) return ptrdiff_t;
    ⋮
    private type Iterator is access all Value_Type;
  end iterators;
end vectors;
```

The above example uses the C interfaces package for unchecked conversion, implementing the above "-" function:

```
function "-"(i: in Iterator; j: in Iterator) return ptrdiff_t is
begin
  return(To_Ptrdiff(To_Addr(i) - To_Addr(j)) / Elmt_Size);
end "-";
```

7 Stream Iterators

Stream iterators in C++ provide an iterator-like interface for the input/output streams. An `istream_iterator` is an input iterator on a given input stream, while an `ostream_iterator` is an output iterator on a given output stream. A special iterator is the `end-of-stream` iterator. Two `end-of-stream` iterators are always equal, a non-`end-of-stream` iterator is always not equal to an `end-of-stream` iterator, while two non-`end-of-stream` iterators are equal if and only if constructed from the same stream.

In terms of our Ada implementation, stream iterators are radically different from other iterators, in that they provide basic functionality, and are not just signatures or adaptors. The stream iterators are instantiated with a type and a procedure to put an element of that type on the stream. In fact, the `Put` operation need not operate only on streams, but can be a simple file operation, since the values placed are all of the same type. In accordance with the C++ STL reference implementation [STL], the output stream iterator increment operator is implemented as a null operation.

A sample output stream iterator specification:

```
generic type value_type is private;
        with procedure Put(v: in value_type);
package ostream_iterators is
  type Iterator is private;
  procedure Inc(i: in out Iterator);
  procedure Assign(i: in Iterator; v: in value_type ;
  private type Iterator is null record;
end ostream_iterators;
```

Here the iterator is a type, defined as a `null record`, which ensures type-safety, while not incurring any overhead since it is purely a compile-time construct.

8 Algorithms

Algorithms, in the Ada implementation of STL, are categorized into packages based on the iterator types they require, with `input_output_algorithms`, `forward_algorithms`, `bidirectional_algorithms` and `random_access_algorithms` all being distinct packages, non-overlapping in functionality. Algorithms are instantiated with iterator signature packages providing the necessary operations.

The algorithms implemented for this study represent a small, but varied, subset of the C++ STL algorithms. Implemented algorithms include `copy`, `find`, `for_each`, `transform`, `reverse`, and `random_shuffle`. The algorithms implemented span all the algorithm classes found in C++ STL.

Following is the specification of `bidirectional_algorithms`:

```
with bidirectional_iterators;
generic with package iterators is new bidirectional_iterators(<>);
        use iterators;
package bidirectional_algorithms is
  procedure reverse( first: in b_iterator; last: in b_iterator );
end bidirectional_algorithms;
```

The implementation of these algorithms raised some unresolved issues in the renaming of overloaded functions. In particular there were several problems with instantiating input/output algorithms with a single iterator type for both input and output. In order to work around these issues some of our functions are in categories different from their C++ counterparts.

9 Adaptors

Only a subset of the C++ adaptors were implemented for this study. One container adaptor, stack, was implemented, but no iterator adaptor. We implemented binders in a manner very similar to that of C++ STL. Negators, however, are an instance of function composition. Function composition is not implemented in C++ STL, but is a good test of the concepts of function adaption and was therefore included in the Ada implementation.

9.1 Stack Adaptors

Stacks are implemented as a signature package, adapting any container with the proper functionality into a stack. This allows algorithms and packages which require stacks to use any implementation through the use of generic stack signature parameters.

9.2 Function Adaptors

Function adaptors in C++ STL make use of inheritance to specify the category of their instantiation function objects. Hence a C++ STL binder only accepts arguments derived from the `binary_function` class. In the Ada implementation this is done using function signatures, which turn out to be particularly handy for this application.

Following is part of the `function_adaptors` package specification, showing the Ada STL specification of `bind2nd` and `unary_composition`.

```
with unary_functions, binary_functions;
package function_adaptors is
  ⋮
  generic with package bound is new binary_functions(<>); use bound;
          second_val: second_argument_type;
  function bind2nd(first_val: first_argument_type) return result_type;
  generic
    with package composition is new unary_composed_functions(<>);
    use composition;
  function unary_composition( x: argument_type ) return result_type;
  ⋮
end function_adaptors;
```

As can be seen above, compositions are instantiated with special composition signatures. This guarantees that the composed functions be of compatible types. This is a good example of how Ada signatures catch conflicts which might pass undetected through a C++compiler.

10 Critique of Ada 95

In this section we will try to present an evaluation of Ada 95 based on our experience in implementing the Standard Template Library, and from the standpoint of two long time C++ users.

We were favorably impressed with the capability of signature packages to represent sets of operations on an abstraction. We also found the strong typing of Ada to be an advantage, although it is important that the facility for unchecked conversions exist. Through explicit instantiation of generic packages and functions the user is aware of the exact parameters used. Generics, as should be expected in a language which has had them since 1983, are cleanly integrated in the language. The `out`-mode in function and procedure parameters is another useful feature of the language. Ada also offers what C++ compilers have yet to standardize on, namely an integrated partial compilation system.

On the negative side, we found the separation of type and parameter passing mode in functions and procedures to be a problem when mapping some of the C++ STL abstractions. The syntactic separation of functions from procedures, along with the lack of a `void` type, restricts generality. Also, the fact that you cannot overload assignment or the index operator prevents Ada STL from ever achieving the same orthogonality as C++ STL. The restriction that generic formal package parameters be instances of a given generic package can repeatedly surprise programmers used to the type freedom of template macros. The need for explicit package and function instantiations can also make for very verbose declaration sections in Ada code.

Our code was compiled with GNAT, the GNU Ada 95 compiler, which is a very capable Ada compiler. The GNAT project is demonstrative of the high level of support Ada users are now getting. We made extensive use of this support while doing our Ada implementation, in particular the GNAT compiler and the "Home of the Brave Ada Programmer" at `http://lglwww.epfl.ch/Ada/`.

11 Conclusion

From this report it should be clear that implementing STL in Ada 95 is quite feasible. The additional support for access types and generic formal package parameters has made Ada as capable as C++, while maintaining the safety of traditional Ada. Our prototype implementation, presented in this report, can, in our opinion, easily be fleshed out into a full STL implementation. This will be pursued by one of the authors as a Master's project under the supervision of Dr. David Musser, and is expected to be completed by May 1996.

To be fair, we have not dealt at all with two of the pillars of the C++ STL design philosophy, i.e., orthogonality and efficiency. Orthogonality with regard to built in types is not easily implemented in Ada, since Ada does not support pointers to array elements in a flexible manner, nor does it allow the overloading of some operations. It is our belief that our prototype implementation can be efficient when compiled with a good compiler. This, however, remains to be proven.

References

[Ada95a] Ada 95 language reference manual. Technical Report ISO/IEC 8652:1995, AD A293760, Intermetrics, Inc., 1995.

[Ada95b] Ada 95 rationale : The language, the standard libraries. Technical Report AD A293708, Intermetrics, Inc., January 1995.

[LRM] Working paper for draft proposed international standard for information systems – programming language C++. Technical Report X3J16/95-0185 WG21/N0785, American National Standards Institute.

[MS89] David R. Musser and Alexander Stepanov. *The Ada generic library : linear list processing packages.* Springer-Verlag, 1989.

[SL94] Alexander Stepanov and Meng Lee. The standard template library. Technical Report HPL-94-34, Hewlett-Packard Laboratories, April 1994.

[STL] The Hewlett-Packard Standard Template Library reference implementation. Hewlett-Packard Laboratories, August, 1994, revised October 31, 1995. `ftp://butler.hpl.hp.com/stl`.

A Ada STL Bidirectional Example

```
with Gnat.IO; use Gnat.IO;

-- Containers
with Lists;

-- Iterators
with Input_Iterators;
with Output_Iterators;
with Bidirectional_Iterators;
with Forward_Iterators;

-- Algorithms
with Input_Output_Algorithms;
with Forward_Algorithms;
with Bidirectional_Algorithms;

-- Other
with Accumulator;
with Ostream_Iterators;

procedure Test_Bidirectional is
  package IL is new Lists(Integer, 0);
  use IL;
  package IL_Iter renames IL.Iterators;
  use IL_Iter;

  procedure My_Put(V : Integer) is
  begin
    Put(V);
    Put(" ");
  end My_Put;

  package OS_Iter is new Ostream_Iterators(Integer, My_Put);
```

```
  use OS_Iter;

  -- Instantiate special iterators (i.e. instantiated differently than in
  --   the container)
  package IL_Output_Iter is
    new Output_Iterators(Integer, OS_Iter.Iterator);

  -- Instantiate required algorithms packages
  package IL_Input_Output_Algo is
    new Input_Output_Algorithms(IL.Input_Iterator, IL_Output_Iter);
  use IL_Input_Output_Algo;
  package IL_Forward_Algo is
    new Forward_Algorithms(IL.Forward_Iterator);
  use IL_Forward_Algo;
  package IL_Bidirectional_Algo is
    new Bidirectional_Algorithms(IL.Bidirectional_Iterator);
  use IL_Bidirectional_Algo;

  -- Instantiate required generic algorithms
  function Find is new IL_Forward_Algo.Find( "=" );
  package IL_Accumulator is new Accumulator(Integer,0,"+");
  procedure Sum_Up is new IL_Forward_Algo.Apply( IL_Accumulator.Add );


  -- Variables
  Cnt : Integer;
  I : IL_Iter.Iterator;
  E : IL_Iter.Iterator;
  L : List;
  OS : OS_Iter.Iterator;

begin
  Put_Line ("Welcome to test_bidirectional !");
  New_Line;
  Put ("Please enter a positive integer now followed by <CR> ");
```

```
Get (Cnt);

Put ("Inserting 1.."); Put (Cnt); Put (" into a list."); New_Line (2);
L := Create;
for I in 1 .. Cnt loop
  Push_Back( L, I );
end loop;
Put( "The list now contains " );
Put( Size(L) );
Put( " elements." );
New_Line;

Put_Line("Printing from Start to Finish.");
OS := Copy(Start(L), Finish(L), OS);
New_Line; New_Line;

Put( "Removing the latter half of the list using pop_back" ); New_Line;
for I in 1 .. Cnt/2 loop
  Pop_Back( L );
end loop;
Put( "The list now contains " ); Put( Size(L) );
Put( " elements." ); New_Line;

Put_Line ("Printing from Start to Finish.");
OS := Copy(Start(L), Finish(L), OS);
New_Line; New_Line;

Put_Line( "Reversing the list" );
IL_Bidirectional_Algo.Reverse_Range( Start(L), Finish(L) );

Put_Line ("Printing from Start to Finish.");
OS := Copy(Start(L), Finish(L), OS);
New_Line; New_Line;

Put( "Looking for " ); Put( Cnt/4 ); Put( " ... " );
```

```
  if Find( Start(L), Finish(L), Cnt/4 ) /= Finish(L) then
    Put_Line( "Found ! (OK)" );
  else
    Put_Line( "Not Found !  (ERROR)" );
  end if;

  Put( "Looking for " ); Put( Cnt ); Put( " ... " );
  if Find( Start(L), Finish(L), Cnt ) /= Finish(L) then
    Put_Line( "Found ! (ERROR)" );
  else
    Put_Line( "Not Found ! (OK)" );
  end if;

  Put( "Accumulating contents of list ... " );
  Sum_Up( Start(L), Finish(L) );
  Put( "Sum is: " ); Put( IL_Accumulator.Sum ); New_Line;

  Put( "Accumulating another pass..." ); New_Line;
  Sum_Up( Start(L), Finish(L) );
  Put( "New sum is: " ); Put( IL_Accumulator.Sum ); New_Line;

  Put ("Printing from Start to Finish.");  New_Line;
  OS := Copy(Start(L), Finish(L), OS);
  New_Line; New_Line;
  New_Line;
  Put_Line( "End of Test_Bidirectional");
end Test_Bidirectional;
```

B Ada STL Random Access Example

```
with Gnat.IO; use Gnat.IO;
with Interfaces.C; use Interfaces.C;

-- Containers
with Vectors;

-- Iterators
with Input_Iterators;
with Output_Iterators;
with Ostream_Iterators;
with Bidirectional_Iterators;
with Forward_Iterators;
with Random_Access_Iterators;

-- Algorithms
with Input_Output_Algorithms;
with Bidirectional_Algorithms;
with Forward_Algorithms;
with Random_Access_Algorithms;

-- Other
with Accumulator;
with Ptrdiff_Random;

procedure Test_Random is
  package IV is new Vectors(Integer);
  use IV;
  package IV_Iter renames IV.Iterators;
  use IV_Iter;

  procedure My_Put(V : Integer) is
  begin
    Put(V);
```

```
  Put(" ");
end My_Put;

package OS_Iter is new Ostream_Iterators(Integer, My_Put);
use OS_Iter;

-- Instantiate all necessary algorithms packages
package IV_Input_Iter is new Input_Iterators
  (Integer, IV_Iter.Iterator);
package IV_Output_Iter is new Output_Iterators
  (Integer, OS_Iter.Iterator);
package IV_InputOutput_Algo is new Input_Output_Algorithms
  (IV_Input_Iter, IV_Output_Iter);
use IV_InputOutput_Algo;
package IV_Forward_Algo is new Forward_Algorithms
  (IV.Forward_Iterator);
use IV_Forward_Algo;
package IV_Bidirectional_Algo is new Bidirectional_Algorithms
  (IV.Bidirectional_Iterator);
use IV_Bidirectional_Algo;
package IV_Random_Access_Algo is new Random_Access_Algorithms
  (IV.Random_Access_Iterator);
use IV_Random_Access_Algo;

-- Instantiate required generic algorithms
function Find is new IV_Forward_Algo.Find( "=" );
package I_Accumulator is new Accumulator(Integer,0,"+");
procedure Sum_Up is new IV_Forward_Algo.Apply( I_Accumulator.Add );

-- Local functions
function My_Random( Gen : Ptrdiff_Random.Generator;
    Max : Ptrdiff_T ) return Ptrdiff_T is
begin
  return(Ptrdiff_Random.Random(Gen) mod Max);
end My_Random;
```

```
  procedure Rshuffle is new IV_Random_Access_Algo.Random_Shuffle( My_Random );

  -- Variables
  Os  : OS_Iter.Iterator;
  Cnt : Integer;
  I   : IV_Iter.Iterator;
  E   : IV_Iter.Iterator;
  V   : Vector;
  Gen: Ptrdiff_Random.Generator;

begin
  Put_Line ("Welcome to Test_Random !");
  New_Line;
  Put ("Please enter a positive integer now followed by <CR> ");
  Get (Cnt);

  Put ("Inserting 1..");  Put (Cnt);  Put (" into a list.");  New_Line (2);
  Create(D=>V);
  for I in 1 .. Cnt loop
    Push_Back( V, I );
  end loop;
  Put( "The vector now contains " );
  Put( Size(V) );
  Put( " elements." );
  New_Line;

  Put_Line("Printing from Start to Finish.");
  Os := Copy(Start(V), Finish(V), Os);
  New_Line; New_Line;

  Put_Line ("Calling random shuffle ..." );
  Ptrdiff_Random.Reset(Gen, 28752341);
  Rshuffle( Start(V), Finish(V), Gen );

  Put ("Printing from Start to Finish.");  New_Line;
```

```
  Os := Copy(Start(V), Finish(V), Os);
  New_Line; New_Line;

  I_Accumulator.Reset;
  Put( "Accumulating contents of vector (after initialization)..." ); New_Line;
  Sum_Up( Start(V), Finish(V) );
  Put( "And the sum is: " ); Put( I_Accumulator.Sum ); New_Line;

end Test_Random;
```

C Ada STL Adaptors Example

```
with Gnat.IO; use Gnat.IO;

with Lists;
with Forward_Algorithms;
with Unary_Functions, Binary_Functions;
with Unary_Composed_Functions, Binary_Composed_Functions;
with Function_Adaptors;
with Stack_Adaptors;

procedure Test_Funcadapts is
  -- Basic operations on lists
  package IL is new Lists(Integer,0); use IL; use IL.Iterators;
  package IL_Forward_Algo is new Forward_Algorithms(IL.Forward_Iterator);
  package IL_Stack is new Stack_Adaptors( IL.List, IL.Size_Type, IL.Value_Type,
  IL.Create, IL.Empty, IL.Size,
  IL.Push_Back, IL.Pop_Back, IL.Back );
  use IL_Forward_Algo;

  -- Define some functions.
  procedure Int_Put( I: Integer ) is
  begin
    Put( I ); Put( " " );
  end;
  function Square( I: Integer ) return Integer is
  begin
    return I*I;
  end;

  -- Some function adaptors
  use Function_Adaptors;
  procedure Put_Range is new Apply( Int_Put );
  package Add_Sig is new Binary_Functions( Integer, Integer, Integer, "+" );
  function Addone is new Bind2nd( Add_Sig, 1 );
```

```
  procedure Inc_Range is new Transform( Addone );
  package Square_Inc_Sig is
    new Unary_Composed_Functions( Integer, Integer, Integer, Addone, Square );
  function Square_Incs is new Unary_Composition( Square_Inc_Sig );
  procedure Inc_Square_Range is new Transform( Square_Incs );

  -- Some variables
  Cnt : Integer;
  L : List;
  I, E : IL.Iterators.Iterator;
  S : IL_Stack.Stack;

begin
  Put_Line ("Hello. Welcome to Test_List" & "!");  New_Line;
  Put ("Please enter a positive integer now followed by <CR> ");
  Get (Cnt);

  Put ("Inserting 1..");  Put (Cnt);  Put (" into a list.");  New_Line;
  L := Create;
  for I in 1 .. Cnt loop
    Push_Back( L, I );
  end loop;
  Put( "The list now contains " ); Put( Size(L) ); Put( " elements." );
  New_Line(2);

  Put ("Output from Start to Finish.");  New_Line;
  Put_Range( Start( L ), Finish( L ) );  New_Line(2);

  Put ("Incrementing all by one.");  New_Line;
  Inc_Range( Start( L ), Finish( L ), Start( L ) );

  Put ("Output again from Start to Finish.");  New_Line;
  Put_Range( Start( L ), Finish( L ) );  New_Line(2);

  Put ("For all x:  x := (x+1)*(x+1).");  New_Line;
```

```
  Inc_Square_Range( Start( L ), Finish( L ), Start( L ) );

  Put ("Output again from Start to Finish.");  New_Line;
  Put_Range( Start( L ), Finish( L ) );  New_Line(2);

  Put ("Pushing the list onto a stack.");  New_Line;
  S := IL_Stack.Create;
  I := Start( L );
  while I /= Finish( L ) loop
     IL_Stack.Push( S, Val( I ) );
     Inc( I );
  end loop;
  Put ("Output the elements from the stack");  New_Line;
  while not IL_Stack.Empty( S ) loop
     Int_Put( IL_Stack.Top( S ) );
     IL_Stack.Pop( S );
  end loop; New_Line;
end;
```