

# CHAPTER IV

## Bootstrapping From Raw Ada

Being able to use Ada's built-in scalar types (i.e., Boolean, Character, Float or Integer) is primarily a matter of convenience. We could, if we desired, create our own package for each of these scalar types. On the other hand, constructing any non-trivial program in Ada usually means using Ada's type constructors (i.e., access, array, and record). This chapter discusses the additional machinery required to use the built-in scalar types in the RESOLVE/Ada discipline. It also discusses how to encapsulate the type constructors so that we can safely use them to implement other components.

The chapter title "Bootstrapping From Raw Ada" should be interpreted in the following sense. Chapter 2 provides principles that permit the component designer to construct locally certifiable components. All these principles assume the following: when layering a new component on top of another component, the other component's interface must have been designed by following the component interface principles of Section 2.1. Therefore, if we want to use Ada's built-in type constructors we must encapsulate them so that their interfaces conform to the principles of Section 2.1. The problem is, we cannot implement these encapsulated components without directly using the built-in type constructors. "Bootstrapping" refers to providing a component whose package specification conforms to the principles of Section 2.1, while its package body does not conform to the principles of Sections 2.2 and 2.3 (i.e., it is programmed in raw Ada). Figure 20 illustrates this by separating concepts for access, array and record types from their implementations, and by placing the implementations in a separate area labeled "Raw Ada":

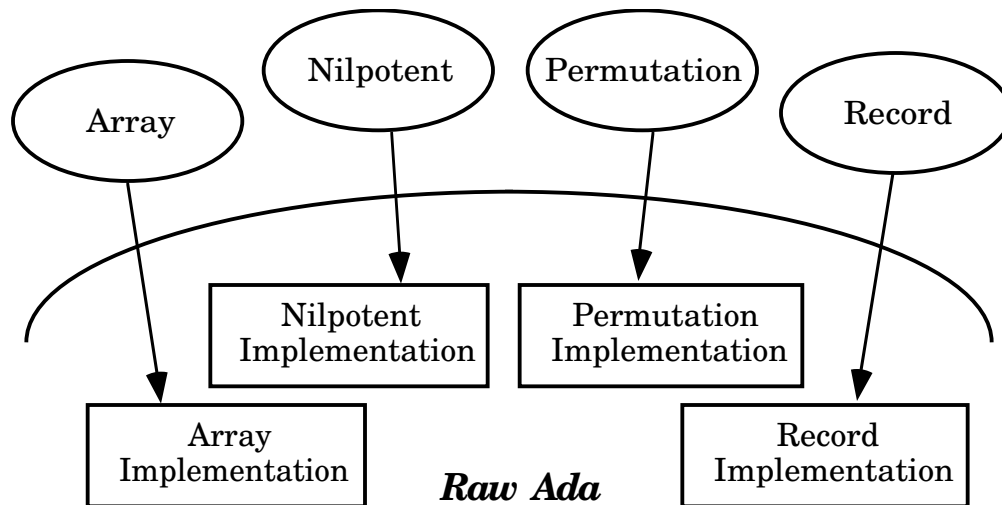


Figure 20 — Bootstrapping From Raw Ada

The remainder of this chapter is organized as follows. Section 4.1 discusses additional operations (e.g., Initialize, Finalize, etc.) required for Ada’s built-in scalar types. Section 4.2 discusses an encapsulation of Ada’s record and array type constructors. Sections 4.3 and 4.4 discuss encapsulations of Ada’s access type constructor.

#### 4.1 Ada’s Built-in Scalar Types

In this section we introduce the `Built_In_Types` package, which eliminates the need to create separate components for providing the types `Boolean`, `Character`, `Integer`, and `Float`. In order to permit seamless composition with other components (designed using the interface principles of Section 2.1), and in order to permit their use in component implementations (designed following principles of Sections 2.2. and 2.3) we must enhance these types with the following operations: `Initialize`, `Finalize`, `Swap`, and `Display`. The package `Built_In_Types` supplies these operations for Ada’s built-in scalar types `Boolean`, `Character`, `Float` and `Integer`. It also exports `Copy` and `Test_If_Equal` operations, which are definable for any type and turn out to be generally useful.

The `Built_In_Types` package eliminates the need to create separate generic packages for `Boolean_Template`, `Character_Template`, `Integer_Template`, and `Real_Template`, and the need to instantiate them, and serves as the interface to (some of) Ada’s scalar types. It does not provide a formal specification for each type, although in principle this behavior could be formally specified [Ogden 91]. This does not present a problem because each type basically adheres to our intuitive notion of what it “should do.” Furthermore, when using these types, there is no possibility of introducing intermodule interference or interprocedural interference (Section 2.4) because none of them maintain abstract module state and none of them permit creating aliases.

The following package provides `Initialize`, `Finalize`, `Swap`, `Copy`, `Test_If_Equal` and `Display` for Ada’s built-in scalar types `Boolean`, `Character`, `Float` and `Integer` (see Appendix B for `Built_In_Types`’ package body):

```
package Built_In_Types is
```

```

-- Boolean -----
procedure Initialize (b: in out Boolean);
--! ensures      "b = FALSE"

procedure Finalize (b: in out Boolean);

procedure Swap (
    b1: in out Boolean;
    b2: in out Boolean);

procedure Copy (
    b1: in out Boolean;           --! preserves
    b2: in out Boolean           --! produces
);
--! ensures      "b2 = b1"

procedure Test_If_Equal (
    b1: in out Boolean;           --! preserves
    b2: in out Boolean;           --! preserves
    equal: in out Boolean        --! produces
);
--! ensures      "equal iff (b1 = b2)"

procedure Display (
    b: in out Boolean;           --! preserves
    indent: in Integer           --! preserves
);
--! requires     "indent >= 0"
--! ensures      "b is output as TRUE or FALSE, starting
--!                indent spaces right of the cursor position at
--!                the time of the call, leaving the cursor position
--!                at the end of the displayed value"

-- Character -----

procedure Initialize (c: in out Character);
--! ensures      "c = ASCII.NUL"

procedure Finalize (c: in out Character);

procedure Swap (
    c1: in out Character;
    c2: in out Character);

procedure Copy (
    c1: in out Character;           --! preserves
    c2: in out Character           --! produces
);
--! ensures      "c2 = c1"

procedure Test_If_Equal (

```

```

        c1: in out Character;           --! preserves
        c2: in out Character;           --! preserves
        equal: in out Boolean           --! produces
    );
--! ensures    "equal iff (c1 = c2)"

procedure Display (
    c: in out Character;                 --! preserves
    indent: in Integer                   --! preserves
);
--! requires    "indent >= 0"
--! ensures    "c is output as an ASCII character, starting
--!            indent spaces right of the cursor position at
--!            the time of the call, leaving the cursor position
--!            at the end of the displayed value"

-- Float -----

procedure Initialize (r: in out Float);
--! ensures    "r = 0.0"

procedure Finalize (r: in out Float);

procedure Swap (
    r1: in out Float;
    r2: in out Float);

procedure Copy (
    r1: in out Float;                     --! preserves
    r2: in out Float                       --! produces
);
--! ensures    "r2 = r1"

procedure Test_If_Equal (
    r1: in out Float;                     --! preserves
    r2: in out Float;                     --! preserves
    equal: in out Boolean                  --! produces
);
--! ensures    "equal iff (r1 = r2)"

procedure Display (
    r: in out Float;                       --! preserves
    indent: in Integer                     --! preserves
);
--! requires    "indent >= 0"
--! ensures    "r is output as a decimal real, starting
--!            indent spaces right of the cursor position at
--!            the time of the call, leaving the cursor position
--!            at the end of the displayed value"

-- Integer -----

procedure Initialize (i: in out Integer);

```

```

--! ensures      "i = 0"

procedure Finalize (i: in out Integer);

procedure Swap (
    i1: in out Integer;
    i2: in out Integer);

procedure Copy (
    i1: in out Integer;           --! preserves
    i2: in out Integer;           --! produces
);
--! ensures      "i2 = i1"

procedure Test_If_Equal (
    i1: in out Integer;           --! preserves
    i2: in out Integer;           --! preserves
    equal: in out Boolean;        --! produces
);
--! ensures      "equal iff (i1 = i2)"

procedure Display (
    i: in out Integer;           --! preserves
    indent: in Integer;          --! preserves
);
--! requires     "indent >= 0"
--! ensures     "i is output as a decimal integer, starting
--!               indent spaces right of the cursor position at
--!               the time of the call, leaving the cursor position
--!               at the end of the displayed value"

end Built_In_Types;

```

## 4.2 Encapsulating Ada's Record and Array Type Constructors

In this section we discuss an encapsulation of Ada's record and array type constructors. The goal is to have record and array components that seamlessly compose with, and can be safely used in an implementation of any of RESOLVE/Ada components. To achieve this, the interfaces of these components must be designed by following the component interface principles of Section 2.1.

This leads to a design for these components that, at first, may seem unusual when compared to Ada's built-in records and arrays. For example, when programming in raw Ada, we move data in and out of a record or an array by using an assignment statement. But the data movement operator for our components is Swap; therefore, our record and array components must move data in and out using Swap. There are other aspects particular to each constructor that we address in the following two sections (e.g., efficiency of exported operations).

## 4.2.1 Ada Records

We begin by looking at an encapsulation for a record with two fields (see Record2\_Template below). By inspection, one can see that this component conforms to the principles of Section 2.1, i.e., it is a formally specified generic package, parameterized by its conceptual types, exporting a limited private type, and Initialize, Finalize, Swap, etc.

```

--! concept Record2_Template

--!      conceptual context

--!      generic
--!      conceptual parameters

--!      type Item1
--!          type Item1 is limited private;
--!          with procedure Initialize (
--!              x: in out Item1);
--!          with procedure Finalize (
--!              x: in out Item1);
--!          with procedure Swap (
--!              x1: in out Item1;
--!              x2: in out Item1);

--!      type Item2
--!          type Item2 is limited private;
--!          with procedure Initialize (
--!              x: in out Item2);
--!          with procedure Finalize (
--!              x: in out Item2);
--!          with procedure Swap (
--!              x1: in out Item2;
--!              x2: in out Item2);

--!      package Record2_Template is
--!      interface

--!          procedure Initialize_Package;
--!          procedure Finalize_Package;

--!      type Record2 is modeled by tuple
--!          field1: math[Item1]
--!          field2: math[Item2]
--!      exemplar r
--!      initialization
--!          ensures "Item1.init (r.field1) and
--!                  Item2.init (r.field2)"
--!      type Record2 is limited private;
--!      procedure Initialize (

```

```

        r: in out Record2);
    procedure Finalize (
        r: in out Record2);
    procedure Swap (
        r1: in out Record2;
        r2: in out Record2);

    procedure Swap_Field1 (
        r: in out Record2;           --! alters
        x: in out Item1              --! alters
    );
    --! ensures      "r.field2 = #r.field2 and
    --!              r.field1 = #x and x = #r.field1"

    procedure Swap_Field2 (
        r: in out Record2;           --! alters
        x: in out Item2              --! alters
    );
    --! ensures      "r.field1 = #r.field1 and
    --!              r.field2 = #x and x = #r.field2"

    private
        type Record2_Rep;
        type Record2 is access Record2_Rep;
    end Record2_Template;
    --! end Record2_Template

```

The following is a client of Record2\_Template. Notice how easily it composes with Queue\_15\_Fixed (see Section 3.3). Also notice that the style that a client programmer must adopt when using records provided by Record2\_Template is a little different since Swap is used to move data in and out of a record instead of assignment.

```

with Built_In_Types; use Built_In_Types;
with Record2_Template;
with Queue_15_Fixed;

procedure Client is
    package Record2_Facility is new
        Record2_Template (
            Integer,
            Initialize,
            Finalize,
            Swap,
            Integer,
            Initialize,
            Finalize,
            Swap
        );

    package Queue_Facility is new

```

```

    Queue_15_Fixed (
        Record2_Facility.Record2,
        Record2_Facility.Initialize,
        Record2_Facility.Finalize,
        Record2_Facility.Swap
    );

use Record2_Facility;
use Queue_Facility;

x: Integer;
r: Record2;
q: Queue;

-----

begin
    Record2_Facility.Initialize_Package;
    Queue_Facility.Initialize_Package;
    Initialize (x);
    Initialize (r);
    Initialize (q);

    x := 7;
    Swap_Field1 (r, x);
    x := 13;
    Swap_Field2 (r, x);
    Enqueue (q, r);

    ...

    Dequeue (q, r);
    Swap_Field1 (r, x);

    ...

    Finalize (q);
    Finalize (r);
    Finalize (x);
    Queue_Facility.Finalize_Package;
    Record2_Facility.Finalize_Package;
end Client;

```

The following is one possible implementation of Record2\_Template. It should be obvious that we are programming in raw Ada since we are violating a number of implementation principles. For example, record variables are declared using Ada's access type, and storage is allocated dynamically when a record is initialized. In this implementation storage for each record is returned to the system heap when a record is finalized. Another implementation might maintain a free list (similar to the implementation of Queue\_11 in Section 2.1.10). Notice that even though we are programming in raw Ada, the implementations of Record2\_Template's Initialize, Finalize, Swap\_Field1 and

Swap\_Field2 use Item1's and Item2's Initialize, Finalize and Swap operations without any problems.

```
with Unchecked_Deallocation;
```

```
package body Record2_Template is
```

```
-----  
--- Declarations -----  
-----
```

```
type Record2_Rep is record  
    field1: Item1;  
    field2: Item2;  
end record;
```

```
-----  
--- Local Operations -----  
-----
```

```
procedure Free is new  
    Unchecked_Deallocation (Record2_Rep, Record2);
```

```
-----  
--- Exported Operations -----  
-----
```

```
procedure Initialize_Package is  
begin  
    null;  
end Initialize_Package;
```

```
procedure Finalize_Package is  
begin  
    null;  
end Finalize_Package;
```

```
-----  
procedure Initialize (  
    r: in out Record2  
    ) is  
begin  
    r := new Record2_Rep;  
    Initialize (r.field1);  
    Initialize (r.field2);  
end Initialize;
```

```
-----  
procedure Finalize (  
-----
```

```

        r: in out Record2
    ) is
begin
    Finalize (r.field2);
    Finalize (r.field1);
    Free (r);
end Finalize;

-----

procedure Swap (
    r1: in out Record2;
    r2: in out Record2
) is
    temp: Record2;
begin
    temp := r1;
    r1 := r2;
    r2 := temp;
end Swap;

-----

procedure Swap_Field1 (
    r: in out Record2;
    x: in out Item1
) is
begin
    Swap (r.field1, x);
end Swap_Field1;

-----

procedure Swap_Field2 (
    r: in out Record2;
    x: in out Item2
) is
begin
    Swap (r.field2, x);
end Swap_Field2;

end Record2_Template;

```

One drawback with encapsulating Ada's record constructor is that separate components must be created for records with different numbers of fields, i.e., we need a `Record3_Template` for records with three fields, a `Record4_Template` for records with four fields, etc. There is no avoiding this since the types (and their standard operations) must be supplied for each of the fields as generic parameters, and there is no way in Ada to have a variable number of generic parameters. Each of these components is similar to `Record2_Template`.

### 4.2.2 Ada Arrays

In this section we discuss the encapsulation of Ada's array constructor. The basic mathematical model used is a function from Integer to Item, where Item is what is stored in the array. Many variations are possible. For example, in this section we present `Static_Array_Template`, so called because the upper and lower bounds are fixed at instantiation time, i.e., all array variables declared from a particular instance have the same bounds. Another variation might allow setting different bounds for each array variable declared.

The following is `Static_Array_Template`'s package specification. One possible implementation, similar to `Record2_Template`'s, can be found in Appendix B.

```
--! concept Static_Array_Template

      generic
--!   conceptual context

--!     conceptual parameters

--!       type Item
          type Item is limited private;
          with procedure Initialize (
            x: in out Item);
          with procedure Finalize (
            x: in out Item);
          with procedure Swap (
            x1: in out Item;
            x2: in out Item);

--!     constants

--!       LOWER_BOUND: integer

--!       UPPER_BOUND: integer

--!     realization context

--!     realization parameters

--!       lower_bound: in Integer;
--!       ensures      "lower_bound = LOWER_BOUND"

--!       upper_bound: in Integer;
--!       ensures      "upper_bound = UPPER_BOUND"

--!     package Static_Array_Template is
--!     interface
```

```

procedure Initialize_Package;
procedure Finalize_Package;

--!
--!   type Static_Array is modeled by
--!       function from integer to math[Item]
--!   exemplar a
--!   constraint "for all i: integer
--!       ((i < LOWER_BOUND or
--!       i > UPPER_BOUND) implies
--!           Item.init (a(i)))"
--!   initialization
--!       ensures "for all i: integer
--!           (Item.init (a(i)))"
--!   type Static_Array is limited private;
--!   procedure Initialize (
--!       a: in out Static_Array);
--!   procedure Finalize (
--!       a: in out Static_Array);
--!   procedure Swap (
--!       a1: in out Static_Array;
--!       a2: in out Static_Array);

procedure Get_Bounds (
--!   a: in out Static_Array;    --! preserves
--!   lower: in out Integer;    --! produces
--!   upper: in out Integer    --! produces
--!   );
--!   ensures      "lower = LOWER_BOUND and
--!                 upper = UPPER_BOUND"

procedure Swap_Entry (
--!   a: in out Static_Array;    --! alters
--!   i: in Integer;            --! preserves
--!   x: in out Item            --! alters
--!   );
--!   requires     "LOWER_BOUND <= i <= UPPER_BOUND"
--!   ensures     "for all j: integer (j /= i implies
--!                 a(j) = #a(j) and
--!                 a(i) = #x and x = #a(i)"

private
--!   type Static_Array_Rep;
--!   type Static_Array is access Static_Array_Rep;
--!   end Static_Array_Template;
--! end Static_Array_Template

```

The implementer of `Static_Array_Template` needs to be concerned with efficiency in at least two areas: data movement (i.e., `Swap`) and initialization and finalization (i.e., `Initialize` and `Finalize`). If done naively, each of these operations could perform in time proportional to the size of the array.

The naive approach to implementing Swap would be to swap the individual items. Fortunately, it is easy to see and implement a constant time Swap operation by swapping a pair of pointers:

```

procedure Swap (
    a1: in out Static_Array;
    a2: in out Static_Array
) is
    temp: Static_Array;
begin
    temp := a1;
    a1 := a2;
    a2 := temp;
end Swap;

```

Implementing constant time Initialize and Finalize operations for an array is possible [Harms 89], but is not so easy. The idea is to amortize the cost of array initialization and finalization over the lifetime of the array. The technique requires a trade (more space for less time) and adds an additional constant factor to the execution time of some operations (e.g., Swap\_Entry). Harms outlines many benefits of implementing constant time Initialize and Finalize operations, one of them being “that the program never takes a ‘linear time lunch break’ ” to do finalization, but rather “takes many ‘constant time snacks’ ” [Harms 89; page 15]. The implementation in Appendix B uses the naive approach to initialization and finalization.

The following client of Static\_Array\_Template illustrates how easy it is to compose it with other components. In this client we create an array of queues of pairs of integers.

```

with Built_In_Types;    use Built_In_Types;
with Record2_Template;
with Queue_15_Fixed;
with Static_Array_Template;

procedure Client is
    package Record2_Facility is new
        Record2_Template (
            Integer,
            Initialize,
            Finalize,
            Swap,
            Integer,
            Initialize,
            Finalize,
            Swap
        );

    package Queue_Facility is new
        Queue_15_Fixed (
            Record2_Facility.Record2,
            Record2_Facility.Initialize,
            Record2_Facility.Finalize,
            Record2_Facility.Swap

```

```

    );

package Array_Facility is new
    Static_Array_Template (
        Queue_Facility.Queue,
        Queue_Facility.Initialize,
        Queue_Facility.Finalize,
        Queue_Facility.Swap,
        1,
        10
    );

use Record2_Facility;
use Queue_Facility;
use Array_Facility;

x: Integer;
r: Record2;
q: Queue;
a: Static_Array;

-----

begin
    Record2_Facility.Initialize_Package;
    Queue_Facility.Initialize_Package;
    Array_Facility.Initialize_Package;
    Initialize (x);
    Initialize (r);
    Initialize (q);
    Initialize (a);

    x := 7;
    Swap_Field1 (r, x);
    x := 13;
    Swap_Field2 (r, x);
    Enqueue (q, r);
    Swap_Entry (a, 1, q);

    ...

    Swap_Entry (a, 1, q);
    Dequeue (q, r);
    Swap_Field1 (r, x);

    ...

    Finalize (a);
    Finalize (q);
    Finalize (r);
    Finalize (x);
    Array_Facility.Finalize_Package;

```

```

        Queue_Facility.Finalize_Package;
        Record2_Facility.Finalize_Package;
end Client;

```

### 4.3 Encapsulating Ada's Access Type Constructor: Acyclic Linked Structures

In this section we discuss `One_Way_Nilpotent_Template`, an encapsulation of Ada's access type constructor. `One_Way_Nilpotent_Template` provides an abstraction for constructing dynamic, acyclic, linear linked structures, useful for implementing “unbounded” one-way lists, stacks, queues, etc. [Hollingsworth 92] also discusses a generalized form of `One_Way_Nilpotent_Template` called `N_Way_Nilpotent_Template` (see Appendix B) which can be used to construct dynamic tree and directed acyclic graph structures. Also, there is an abstraction available for constructing cyclic linked structures called `Permutation_Template` which is introduced in the following section (Section 4.4).

An appealing aspect about these components is that all their operations can be implemented to execute in constant time. Furthermore, they can reclaim storage automatically. This is especially important since all of our unbounded components — components implemented using dynamically allocated data structures — are implemented using `Permutation_Template`, `One_Way_Nilpotent_Template`, and `N_Way_Nilpotent_Template`. Clients of these three components need not be concerned with storage management; indeed they *may not* be if they observe the client implementation principles. The remainder of this section describes `One_Way_Nilpotent_Template`'s package specification, and possible implementations, following the lines of the presentation in [Hollingsworth 92].

#### 4.3.1 Intuition Behind `One_Way_Nilpotent_Template` and its Specification

`One_Way_Nilpotent_Template` can be used to implement other unbounded components (e.g., one-way list, queue, stack, etc.). Without it, each of these other components (e.g., queue) would have to be implemented in raw Ada. Thus, `One_Way_Nilpotent_Template` provides an opportunity for us to get pointers “right” for acyclic linked structures once and for all down in raw Ada. Furthermore, it eliminates the troublesome details associated with using programming language pointers when implementing other unbounded components.

Experience shows that the abstraction may be difficult to understand, so we begin with a simple example, working up from there. Suppose one is developing a program that requires a singly linked list for storing a character string. There is one problem: the programming language being used does not support pointers. What can be done? Simulate the pointers using an array for storing the characters while maintaining a parallel array for storing the simulated pointer link to the next character. This is common in FORTRAN programs, and is taught in some introductory data structures courses and standard texts; see [Horowitz 76]. For example, suppose we have the list (a, x, r). An implementation using simulated pointers and parallel arrays might look like the following:

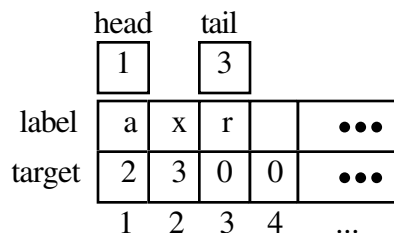


Figure 21 — Simulated Pointers Using Parallel Arrays  
For a Singly Linked Structure

To access the first item in the list, “a,” use the value stored in the variable head to index into the label array. To find the next item in the list, simply index into the target array and use the value stored there as the index into the label array. The end of the list is reached when the value in the target array is zero.

Abstracting from this implementation leads to a specification for the desired generic abstract data type. There are actually two mappings at work in this example, a mapping from integers to characters, and a mapping from integers to integers. These mappings can be viewed as mathematical functions, label and target, having the following mathematical form:

label: integer  $\rightarrow$  Item  
target: integer  $\rightarrow$  integer

By definition, label and target are both total functions that form part of a complete mathematical model of the simulated pointer structure. In examples we show only ordered pairs for which the domain value is of interest. For the above example, the functions have the following values:

label =     {(1, a), (2, x), (3, r)}  
target = {(1, 2), (2, 3), (3, 0)}

`One_Way_Nilpotent_Template`, presented below, is based on the two functions label and target. It exports a program type called Position (modeled by the integers used as the domains of the functions), and it exports operations for manipulating the functions (i.e., for evaluating and changing them). The package is parameterized by the type Item.

We have some experience introducing `One_Way_Nilpotent_Template` to programmers in the classroom (eighteen graduate and upper-division undergraduate students, in a class called “Software Components Using Ada”; see [Hollingsworth 91]). The students’ first encounter with `One_Way_Nilpotent_Template` was difficult. However, with some explanation of the specs along with an example similar to the one found later in the section, the students became comfortable with the abstraction and easily were able to use it to implement two different linked structure packages (queue and one-way list).

```
--! concept One_Way_Nilpotent_Template

--!      generic
--!      conceptual context
```

```

--!      conceptual parameters
--!
--!      type Item
--!          type Item is limited private;
--!          with procedure Initialize (
--!              x: in out Item);
--!          with procedure Finalize (
--!              x: in out Item);
--!          with procedure Swap (
--!              x1: in out Item;
--!              x2: in out Item);
--!
--!      variables
--!
--!          used: integer
--!          label: function from integer to math[Item]
--!          target: function from integer to integer
--!
--!      initially "used = 0 and
--!          for all i: integer
--!          (Item.init (label (i)) and (target (i) = 0))"
--!
--!      package One_Way_Nilpotent_Template is
--!      interface
--!
--!          procedure Initialize_Package;
--!          procedure Finalize_Package;
--!
--!      type Position is modeled by integer
--!      exemplar p
--!      constraint "p >= 0"
--!      initialization
--!          ensures "p = 0"
--!      type Position is limited private;
--!      procedure Initialize (
--!          p: in out Position);
--!      procedure Finalize (
--!          p: in out Position);
--!      procedure Swap (
--!          p1: in out Position;
--!          p2: in out Position);
--!
--!      procedure Label_New_Position (
--!          p: in out Position;          --! produces
--!          x: in out Item              --! consumes
--!      );
--!      other affected variables
--!          alters used
--!          alters label
--!      ensures "used = #used + 1 and p = used and
--!          for all i: integer (i /= p implies
--!              label (i) = #label (i)) and
--!              label (p) = #x"

```

```

procedure Swap_Label (
    p: in out Position;          --! preserves
    x: in out Item              --! alters
);
--! other affected variables
--! alters label
--! requires "p /= 0"
--! ensures "for all i: integer (i /= p implies
--! label (i) = #label (i)) and
--! label (p) = #x and x = #label (p)"

procedure Apply_Target (
    p: in out Position          --! alters
);
--! requires "p /= 0"
--! ensures "p = target (#p)"

procedure Change_Target (
    p1: in out Position;        --! preserves
    p2: in out Position        --! preserves
);
--! other affected variables
--! alters target
--! requires "p1 /= 0 and p1 /= p2 and
--! there does not exist k: integer,
--! (k ≥ 0 and (targetk(p2) = p1))
--! ensures "for all i: integer (i /= p1 implies
--! target (i) = #target (i)) and
--! target (p1) = p2"

procedure Copy (
    p1: in out Position;        --! preserves
    p2: in out Position        --! produces
);
--! ensures "p2 = p1"

procedure Test_If_Equal (
    p1: in out Position;        --! preserves
    p2: in out Position;        --! preserves
    equal: in out Boolean      --! produces
);
--! ensures "equal iff (p1 = p2)"

private
type Position_Rep;
type Position is access Position_Rep;
end One_Way_Nilpotent_Template;
--! end One_Way_Nilpotent_Template

```

The following provides an English explanation for `One_Way_Nilpotent_Template`'s package specification:

- *The mathematical variables.* `One_Way_Nilpotent_Template`'s “conceptual context” introduces three internal “state” variables, `used`, `label` and `target`. The exported operations manipulate these variables as well as their actual parameters. `One_Way_Nilpotent_Template`'s implementation (package body) is *not obligated* to represent these variables explicitly, as they are mathematical abstractions, *not* Ada variables.
- *The exported type `Position` and `Label_New_Position` operation.* Conceptually the type `Position` is modeled by a mathematical integer. Every variable `p` of type `Position` is initially 0. Unused positions are dispensed to the client one at a time, beginning at integer number one, via the operation `Label_New_Position`. Each time a position is dispensed, the math variable `used` is incremented by one. There is no danger of eventual overflow because `used` is a mathematical integer, not an Ada integer. `Label_New_Position` alters the label function, mapping the new `Position` `p` to the Item `x`. The new `Position`'s target is 0 because `target` initially maps every integer to 0.
- *The `Swap_Label` operation.* Conceptually, `Swap_Label` allows a client to change the label function at `Position` `p` and simultaneously to obtain the former label at `Position` `p`, by swapping. Neither `used` nor `target` is changed.
- *The `Apply_Target` operation.* Conceptually, this operation allows a client to apply the target function to `p` and sets `p` to the value produced by the application. None of the internal mathematical variables are changed.
- *The `Change_Target` operation.* Conceptually this operation allows a client to alter the target function by changing `target` (`p1`), i.e., `p1` now maps to `p2` under the target function. Neither `used` nor `label` is changed. Note: The notation  $\text{target}^k(p2)$  denotes the iterated application of `target` to `p2`, `k` times. For example,  $\text{target}^2(p2) = \text{target}(\text{target}(p2))$ . The requires clause must be met so that no circular structures will be created. Thus, for any position, applying the target function sufficiently often must eventually end up at position 0. In other words, `target` is a nilpotent function; hence the name of the package.
- *The `Copy` operation.* Conceptually this operation allows a client to create a copy of a position `p1`. None of the internal mathematical variables are changed. Note: This is similar to aliasing if pointers were being used. Assignment is not available for limited private types. Implementations of `One_Way_Nilpotent_Template` can take advantage of this situation so that dangling references and storage leaks are never created. Furthermore, client programmers can take advantage of it by being able to reason locally about copies of positions. Recall from Section 2.2.3 the discussion concerning explicit aliasing and how it can jeopardize local reasoning about correctness. Local reasoning is not jeopardized because `One_Way_Nilpotent_Template` (and `Permutation_Template` introduced in Section 4.4) provide the mathematical machinery for client operations to capture in their ensures clause all effects produced by the operation. In other words, the ensures clause of an operation `P`, which is a client of `One_Way_Nilpotent_Template` (or `Permutation_Template`), captures all changes produced by `P` even if it is called with an alias. This is done by including references in the ensures clause to `One_Way_Nilpotent_Template`'s `target` and `label` variables whenever `P` makes a change to these variables. Because of this, any client of `P` can be assured that nothing

other than the variables explicitly identified in its postcondition is affected, i.e., any client of  $P$  can be assured “and nothing else changes.”

- *The Test\_If\_Equal operation.* Conceptually this operation sets equal to true if and only if  $p1$  and  $p2$  are the same integer. None of the internal mathematical variables are changed.

#### 4.3.2 *Queue\_Template Implemented Using One\_Way\_Nilpotent\_Template*

Next we examine a client of `One_Way_Nilpotent_Template`. In this example, we implement our `Queue_Template` concept using `One_Way_Nilpotent_Template` and `Record2_Template` (see Section 4.2.1). This is yet another possible implementation of our `Queue_Template`, i.e., the only difference between this package specification and `Queue_15`'s package specification is its name and its realization context. Therefore, the following component can be substituted in a client for `Queue_15` without any changes to the client's executable code.

```
--! concept Queue_Template

    generic
--!     conceptual context

--!     uses
--!         STRING_THEORY_TEMPLATE

--!     conceptual parameters

--!     type Item
--!         type Item is limited private;
--!         with procedure Initialize (
--!             x: in out Item);
--!         with procedure Finalize (
--!             x: in out Item);
--!         with procedure Swap (
--!             x1: in out Item;
--!             x2: in out Item);

--!     mathematics

--!     math facilities
--!         STRING_THEORY is
--!             STRING_THEORY_TEMPLATE (math[Item])

--!     realization context

--!     uses
--!         One_Way_Nilpotent_Template, Record2_Template

--!     realization parameters

--!     facility One_Way_Nilpotent_Facility is
```

```

--!
    One_Way_Nilpotent_Template (Item)
type Position is limited private;
with procedure Initialize (
    p: in out Position);
with procedure Finalize (
    p: in out Position);
with procedure Swap (
    p1: in out Position;
    p2: in out Position);
with procedure Label_New_Position (
    p: in out Position;
    x: in out Item);
with procedure Swap_Label (
    p: in out Position;
    x: in out Item);
with procedure Apply_Target (
    p: in out Position);
with procedure Change_Target (
    p1: in out Position;
    p2: in out Position);
with procedure Copy (
    p1: in out Position;
    p2: in out Position);
with procedure Test_If_Equal (
    p1: in out Position;
    p2: in out Position;
    equal: in out Boolean);

--!
facility Record2_Facility is
--!
    Record2_Template (Position, Position)
type Record2 is limited private;
with procedure Initialize (
    r: in out Record2);
with procedure Finalize (
    r: in out Record2);
with procedure Swap (
    r1: in out Record2;
    r2: in out Record2);
with procedure Swap_Field1 (
    r: in out Record2;
    x: in out Position);
with procedure Swap_Field2 (
    r: in out Record2;
    x: in out Position);

package Queue_16 is
--!
interface

    procedure Initialize_Package;
    procedure Finalize_Package;

```

```

--!      type Queue is modeled by STRING
--!      exemplar q
--!      initialization
--!      ensures "q = EMPTY"
      type Queue is limited private;
      procedure Initialize (
        q: in out Queue);
      procedure Finalize (
        q: in out Queue);
      procedure Swap (
        q1: in out Queue;
        q2: in out Queue);

      procedure Enqueue (
        q: in out Queue;           --! alters
        x: in out Item             --! consumes
      );
--!      ensures      "q = APPEND (#q, #x)"

      procedure Dequeue (
        q: in out Queue;           --! alters
        x: in out Item             --! produces
      );
--!      requires     "q /= EMPTY"
--!      ensures     "PREPEND (q, x) = #q"

      procedure Test_If_Empty (
        q: in out Queue;           --! preserves
        empty: in out Boolean     --! produces
      );
--!      ensures     "empty iff q = EMPTY"

      private
      type Queue is record
        rep: Record2;
--!      adjunct length: Integer;
      end record;
      end Queue_16;
--! end Queue_Template

```

Note the following items concerning the implementation of Queue\_16 presented below:

- It always maintains a pre-front dummy position in the linked structure representation of its Queue variables. This is a module-level convention that all operations must observe (see Section 2.3.2). This allows Enqueue and Dequeue to be implemented without special-case code for an empty Queue, because there is always at least one position in the underlying linked structure. When there are items in the Queue, the dummy pre-front is always located at the front of the linked structure, and the first item to be dequeued is its successor.

- An adjunct variable is maintained (see Section 2.3.2). The adjunct variable “length” is declared as a formal comment in the private part of Queue\_16’s package specification above, and is updated in formal comments in Queue\_16’s package body. It is used to simplify the specification of Queue\_16’s convention and correspondence formal comments.
- The specification of the convention and correspondence depends on additional mathematical machinery which is introduced as conceptual context. This additional conceptual context augments the conceptual context introduced in Queue\_16’s package specification. The additional functions introduced, Product and Theta, are used to specify that q (i.e., string of Items) can be obtained by iterating the target function and appending together the labels of each of the positions encountered during the iteration.
- The “renames” declaration is used to rename Record2\_Template’s Swap\_Field1 and Swap\_Field2 operations. This is done to aid clarity.

```

package body Queue_16 is

  --! conceptual context

  --! functions

  --! Product yields s: STRING
  --! parameters
  --! begin: integer
  --! end: integer
  --! f: function from integer to math[Item]
  --! "(end < begin implies
  --! product (begin, end, f) = EMPTY) and
  --! (begin <= end implies
  --! Product (begin, end, f) =
  --! APPEND (Product (begin, end - 1, f),
  --! f (end)))"

  --! Theta yields f: function from integer
  --! to math[Item]
  --! parameters
  --! p: integer
  --! "for all n: integer
  --! ((Theta (p)) (n)) = label (target ^ n (p)))"

  -----
  --- Declarations -----
  -----

  --! type Queue
  --! convention
  --! "0 <= q.length and
  --! target^q.length (field1 (q.rep)) = field2 (q.rep) and
  --! field1 (q.rep) /= 0"
  --! correspondence
  --! "q = Product (1, q.length, Theta (field1 (q.rep)))"

  -----

```

```

--- Local Operations -----
-----

procedure Swap_Pre_Front (
    r: in out Record2;
    x: in out Position
) renames Swap_Field1;

-----

procedure Swap_Rear (
    r: in out Record2;
    x: in out Position
) renames Swap_Field2;

-----

--- Exported Operations -----
-----

procedure Initialize_Package is
begin
    null;
end Initialize_Package;

-----

procedure Finalize_Package is
begin
    null;
end Finalize_Package;

-----

procedure Initialize (
    q: in out Queue
) is
    pre_front: Position;
    rear: Position;
    x: Item;
begin
    Initialize (q.rep);

        Initialize (pre_front);
        Initialize (rear);
        Initialize (x);

    -- Get dummy "pre_front" position
    Label_New_Position (pre_front, x);

    -- Make dummy position the pre_front and
    -- rear of queue
    Copy (pre_front, rear);
    Swap_Pre_Front (q.rep, pre_front);
    Swap_Rear (q.rep, rear);

```

```

--!      q.length := 0;

          Finalize (x);
          Finalize (rear);
          Finalize (pre_front);
end Initialize;

-----

procedure Finalize (
    q: in out Queue
) is
begin
    Finalize (q.rep);
end Finalize;

-----

procedure Swap (
    q1: in out Queue;
    q2: in out Queue
) is
begin
    Swap (q1.rep, q2.rep);
--!      Swap (q1.length, q2.length);
end Swap;

-----

procedure Enqueue (
    q: in out Queue;
    x: in out Item
) is
    rear: Position;
    new_rear: Position;
begin
    Initialize (rear);
    Initialize (new_rear);

    Swap_Rear (q.rep, rear);

    -- Create new position with x as its label
    Label_New_Position (new_rear, x);

    -- Make new position the rear position of the queue
    Change_Target (rear, new_rear);
    Apply_Target (rear);

    Swap_Rear (q.rep, rear);
--!      q.length := q.length + 1;

    Finalize (new_rear);
    Finalize (rear);
end Enqueue;

```

```
-----  
procedure Dequeue (  
    q: in out Queue;  
    x: in out Item  
    ) is  
    pre_front: Position;  
begin  
    Initialize (pre_front);  
  
    Swap_Pre_Front (q.rep, pre_front);  
  
    -- Make successor the new pre_front position  
    -- Swap x out from pre-front's successor position  
    Apply_Target (pre_front);  
    Swap_Label (pre_front, x);  
  
    Swap_Pre_Front (q.rep, pre_front);  
--!    q.length := q.length - 1;  
  
    Finalize (pre_front);  
end Dequeue;  
  
-----  
  
procedure Test_If_Empty (  
    q: in out Queue;  
    empty: in out Boolean  
    ) is  
    pre_front: Position;  
    rear: Position;  
begin  
    Initialize (pre_front);  
    Initialize (rear);  
  
    Swap_Pre_Front (q.rep, pre_front);  
    Swap_Rear (q.rep, rear);  
  
    Test_If_Equal (pre_front, rear, empty);  
  
    Swap_Rear (q.rep, rear);  
    Swap_Pre_Front (q.rep, pre_front);  
  
    Finalize (rear);  
    Finalize (pre_front);  
end Test_If_Empty;  
  
end Queue_16;
```

### 4.3.3 Alternative Implementations for One\_Way\_Nilpotent\_Template

Before discussing implementations for `One_Way_Nilpotent_Template`, we note the fundamental properties shared by all alternatives. Remember that `One_Way_Nilpotent_Template` has been designed for building acyclic linked structures. Consequently, the precondition for `Change_Target` requires that no circularity be introduced into the linked structure that is under construction. This requirement allows implementations to maintain reference counts for all dynamically allocated storage; see [Weide 86].

For example, when the client invokes `Label_New_Position`, storage is allocated for a new position, and its reference count is set to one. The counts are updated by all operations of the `One_Way_Nilpotent_Template` that change the values of `Position` variables or the target function. When the count reaches zero (note that the count can be decremented by a call to `Finalize`, `Label_New_Position`, `Change_Target`, `Apply_Target`, or `Copy`) there are no positions that have access to the allocated storage. At this time, the allocated storage may be reclaimed. Because of its interface, `One_Way_Nilpotent_Template` has complete control over aliasing of allocated storage. If it did not, then the reference count system would break down.

This is also why circular linked structures are not allowed by the `One_Way_Nilpotent_Template` as it stands. If they are allowed (by removing the precondition of `Change_Target`), it is possible to build a structure where each piece of allocated storage has a reference count equal to one, but no position has access to the structure. To detect this situation, the implementation has to follow the target chain of a position whenever a reference count is decremented; otherwise storage leaks might occur. Following the target chain is potentially inefficient. To build circular linked structures (and to have all operations perform in constant time), we have a different abstract data type (see `Permutation_Template` introduced in the following section).

Here are four possible alternative strategies for storage reclamation:

- 1) Use the underlying run time system's garbage collector.
- 2) Use `Unchecked_Deallocation`.
- 3) Maintain an internal free list of individual pieces of storage allocated by `Label_New_Position`, as shown:

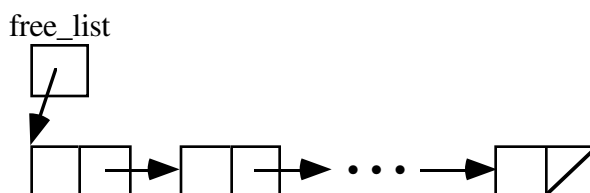


Figure 22 — Free List of Individual Pieces of Storage

- 4) Maintain an internal free list of freed linked structures. These structures have been constructed using `Label_New_Position` and `Change_Target`, and have then been freed:

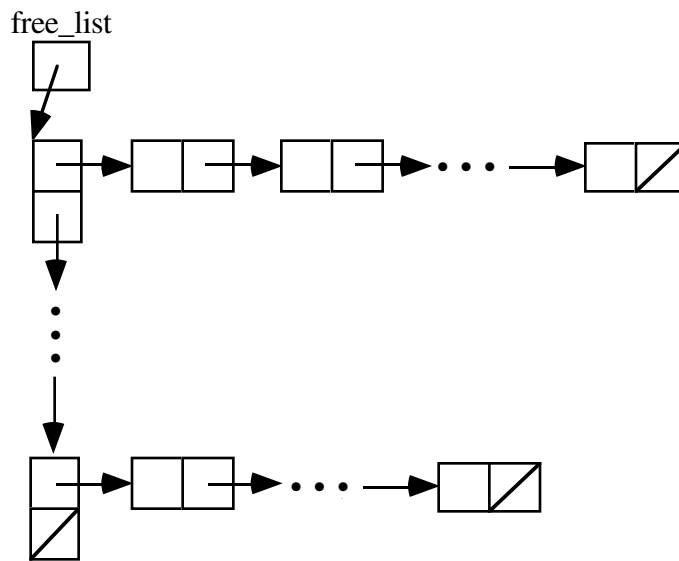


Figure 23 — Free List of Linked Structures

The first two strategies are straightforward, not warranting further discussion; an implementation using the second strategy can be found in Appendix B. The third strategy performs well when individual pieces of storage are freed (e.g., when a stack is popped), but suffers when an entire linked structure is freed. For example, if a client finalizes a stack of size  $N$ , this alternative's performance is necessarily linear in  $N$  (the second strategy has this problem also). Why? Because it has to take each piece of storage off the stack and place it onto the free list. This might lead one to believe that the Finalize operation for a linked structure is inherently a linear-time operation, but it is not. The fourth strategy accommodates a constant time Finalize operation for linked structures of length  $N$ ; see [Weizenbaum 63], [Weide 86] and [Pittel 90]. As stated above, this implementation maintains a free list of freed linked structures rather than a free list of individually allocated pieces of storage. The Finalize operation simply adds the entire size  $N$  linked structure to the free list, in constant time. With this storage management strategy, all `One_Way_Nilpotent_Template` operations take constant time.

#### 4.4 Encapsulating Ada's Access Type Constructor: Cyclic Linked Structures

In the previous section we introduced `One_Way_Nilpotent_Template`, an encapsulation of Ada's access type that can be used to build acyclic linked structures. This section introduces `Permutation_Template`, which is an encapsulation of Ada's access type that can be used to build cyclic linked structures.

#### 4.4.1 Intuition Behind *Permutation\_Template* and its Specification

*One\_Way\_Nilpotent\_Template* is based on the properties of a nilpotent function. If a function  $f$  is nilpotent, then there is a value  $z$  such that the repeated iteration of  $f$  on any value from its domain eventually produces  $z$ . In this section, we introduce another encapsulation for building cyclic linked structures called the *Permutation\_Template*. If a function  $f$  is a permutation function, then it is one-to-one and onto, and repeated iteration of  $f$  on any value from its domain eventually produces the value used to start the iteration. Another property of a permutation function is that it has a well-defined inverse. For example, the following set of ordered pairs is a permutation function (also shown is  $f$ 's inverse):

$$f = \{(1, 3), (3, 7), (7, 10), (10, 1)\}$$

$$f^{-1} = \{(1, 10), (3, 1), (7, 3), (10, 7)\}$$

No matter which initial value from  $f$ 's domain we start from, if we iterate  $f$  (or  $f$ 's inverse) four times, we will return to that value; e.g.,  $f(3) = 7$ ,  $f(7) = 10$ ,  $f(10) = 1$ , and  $f(1) = 3$ .

To introduce *One\_Way\_Nilpotent\_Template*, we demonstrated how a singly linked list (implemented with pointers) can be simulated by using parallel arrays in the obvious way. We can do the same for a circular doubly linked list. For example, suppose we have the list  $(a, x, r)$ . An implementation using simulated pointers and parallel arrays might look like the following, where the forward and backward links are called “permutation” and “permutation inverse” to demonstrate how we can use a permutation function to model cyclic doubly linked structures.

	current position				
	1				
label	a	x	r		...
permutation	2	3	1		...
permutation inverse	3	1	2		...
	1	2	3	4	...

Figure 24 — Simulated Pointers for Doubly Linked Structure

The following is the package specification for *Permutation\_Template*:

```

--! concept Permutation_Template

--!   generic
--!   conceptual context

--!   conceptual parameters

--!   type Item

```

```

    type Item is limited private;
    with procedure Initialize (
        x: in out Item);
    with procedure Finalize (
        x: in out Item);
    with procedure Swap (
        x1: in out Item;
        x2: in out Item);

--!
--!     variables
--!         used: integer
--!         label: function from integer to math[Item]
--!         permutation: function from integer to integer

--!
--!     constraint "used >= 0 and
--!         for all i: integer
--!             ((i <= 0 or i > used) implies
--!                 (Item.init (label (i)) and
--!                     permutation (i) = i)) and
--!         for all i: integer (there exists k: integer
--!             (k >= 1 and permutation^k (i) = i))"

--!
--!     initially "used = 0"

package Permutation_Template is
interface

    procedure Initialize_Package;
    procedure Finalize_Package;

--!
--!     type Position is modeled by integer
--!     exemplar p
--!     constraint "p >= 0"
--!     initialization
--!         other affected variables
--!         alters used
--!     ensures "used = #used + 1 and p = used"
    type Position is limited private;
    procedure Initialize (
        p: in out Position);
    procedure Finalize (
        p: in out Position);
    procedure Swap (
        p1: in out Position;
        p2: in out Position);

    procedure Label_New_Position (
        p: in out Position;           --! produces
        x: in out Item                --! consumes
    );
--!
--!     other affected variables

```

```

--!           alters used
--!           alters label
--!     ensures   "used = #used + 1 and p = used and
--!               for all i: integer (i /= p implies
--!                 label (i) = #label (i)) and
--!                 label (p) = #x"

procedure Swap_Label (
  p: in out Position;           --! preserves
  x: in out Item                --! alters
);
--!   other affected variables
--!   alters label
--!   ensures   "for all i: integer (i /= p implies
--!             label (i) = #label (i)) and
--!             label (p) = #x and x = #label (p)"

procedure Apply_Permutation (
  p: in out Position           --! alters
);
--!   ensures   "p = permutation (#p)"

procedure Apply_Inverse_Permutation (
  p: in out Position           --! alters
);
--!   ensures   "#p = permutation (p)"

procedure Transpose_Before (
  p1: in out Position;         --! preserves
  p2: in out Position         --! preserves
);
--!   other affected variables
--!   alters permutation
--!   ensures   "for all i: integer
--!             (i /= p1 and i /= p2 implies
--!               permutation-1 (i) =
--!               #permutation-1 (i))
--!             and
--!             permutation-1 (p1) = #permutation-1 (p2)
--!             and
--!             permutation-1 (p2) = #permutation-1 (p1)"

procedure Transpose_After (
  p1: in out Position;         --! preserves
  p2: in out Position         --! preserves
);
--!   other affected variables
--!   alters permutation
--!   ensures   "for all i: integer
--!             (i /= p1 and i /= p2 implies
--!               permutation (i) = #permutation (i))
--!             and

```

```

--!           permutation (p1) = #permutation (p2)
--!           and
--!           permutation (p2) = #permutation (p1)"

procedure Copy (
    p1: in out Position;           --! preserves
    p2: in out Position           --! produces
);
--! ensures      "p2 = p1"

procedure Test_If_Equal (
    p1: in out Position;           --! preserves
    p2: in out Position;           --! preserves
    equal: in out Boolean         --! produces
);
--! ensures      "equal iff (p1 = p2)"

private
    type Position_Rep;
    type Position is access Position_Rep;
end Permutation_Template;
--! end Permutation_Template

```

The `Permutation_Template` operations `Label_New_Position`, `Swap_Label`, `Copy` and `Test_If_Equal` provide essentially the same function as the corresponding operations provided by `One_Way_Nilpotent_Template`; see Section 4.3.1's explanation of these operations. `Permutation_Template`'s `Apply_Permutation` and `Apply_Inverse_Permutation` are similar to `One_Way_Nilpotent_Template`'s `Apply_Target`. While `Permutation_Template`'s `Transpose_Before` and `Transpose_After` are similar to `One_Way_Nilpotent_Template`'s `Change_Target`, there is enough difference to warrant further explanation.

`One_Way_Nilpotent_Template` exports the operation `Change_Target` to allow a client to alter the target function. This operation permits the client to “link” `One_Way_Nilpotent` positions and thereby construct linked structures. `Permutation_Template` exports two operations called `Transpose_Before` and `Transpose_After` that permit the client to link `Permutation` positions by altering the permutation function. We illustrate the use of these two functions with a number of examples.

*First example, linking and unlinking two cycles.* This example uses a client program to illustrate the linking of two previously unlinked positions each contained in its own cycle. After linking the two unlinked cycles, the client program then unlinks them, producing the original cycles. The client program might look like the following:

```

begin
    Initialize (p1);
    Initialize (p2);

    Transpose_After (p1, p2);
    Transpose_After (p1, p2);

```

```

Finalize (p2);
Finalize (p1);
end;

```

The client program creates two positions each in its own cycle. The following diagram illustrates this situation just after initialization and prior to the first `Transpose_After`:

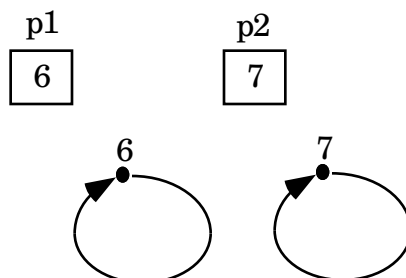


Figure 25 — Two Positions in Their Own Cycle

After executing `Transpose_After`, there is only one cycle containing both positions:

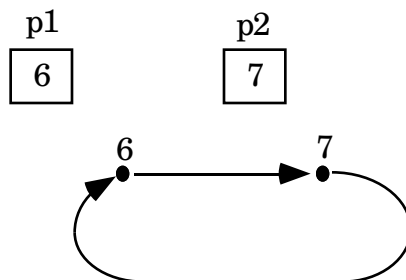


Figure 26 — Two Positions Linked by One Cycle

Finally the original two cycles are recovered by the client program when it executes the second `Transpose_After`; see Figure 25 above.

*Second example, removing a position from a cycle.* This example illustrates how to remove a position from a cycle containing more than two positions. The following cycle contains six positions:

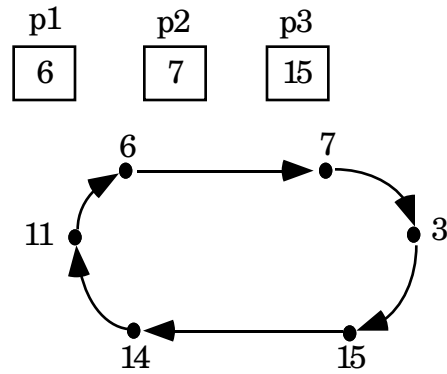
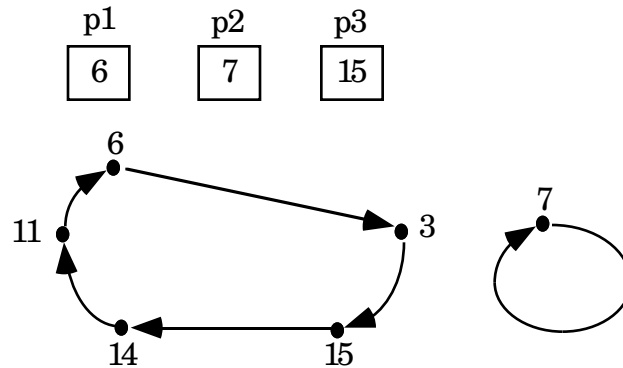
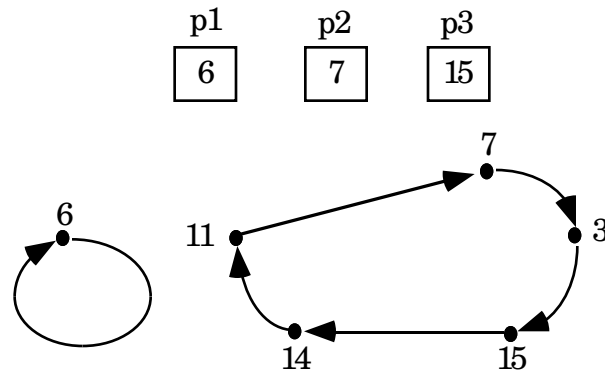


Figure 27 — Cycle Containing More Than Two Positions

The following figures (Figures 28 and 29) illustrate, respectively, the result of executing a `Transpose_After` (`p1`, `p2`) or a `Transpose_Before` (`p1`, `p2`).

Figure 28 — One Position Removed From a Cycle Using `Transpose_After`Figure 29 — One Position Removed From a Cycle Using `Transpose_Before`

*Third example, splitting a cycle and joining two cycles.* This example illustrates how to split a cycle containing more than two positions. For this example, we use the cycle given

in Figure 27 above and execute `Transpose_After (p2, p3)`. The result is illustrated in Figure 30.

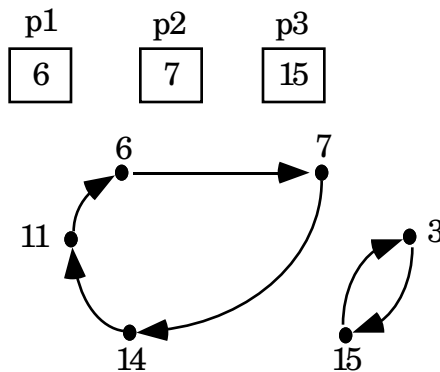


Figure 30 — Splitting a Cycle into Two

Finally, we can join the two cycles illustrated in Figure 30 by executing `Transpose_After (p2, p3)`, giving us the original cycle found in Figure 27. Or, to again illustrate the use of `Transpose_Before`, we might join them by executing `Transpose_Before (p1, p3)`, giving the cycle in Figure 31.

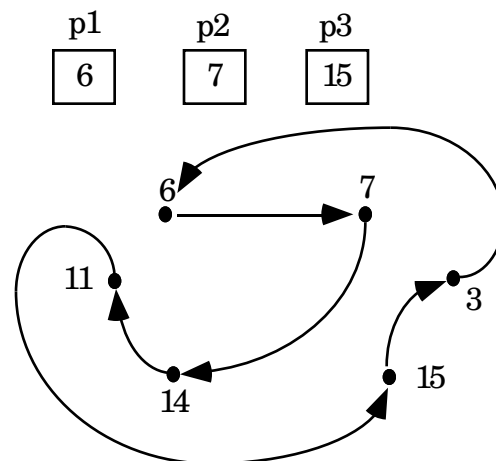


Figure 31 — Joining Two Cycles

#### 4.4.2 *Two\_Way\_List\_Template Implemented Using Permutation\_Template*

Next we demonstrate the application of `Permutation_Template` by using it to implement `Two_Way_List_Template`. Conceptually, `Two_Way_List_Template` is the same as `One_Way_List_Template`, except that the former also exports the operation `Retreat`. `Retreat` permits moving the fence toward the start of the list. By using `Permutation_Template`, we can implement `Advance` and `Retreat` to work in constant time. `Two_Way_List_Template`'s package specification can be found in Appendix A. Its package body follows:

```

package body Two_Way_List_Template is

--! conceptual context

--! functions

--! Product yields s: STRING
--! parameters
--!   begin: integer
--!   end: integer
--!   f: function from integer to math[Item]
--!   "(end < begin implies
--!     product (begin, end, f) = EMPTY) and
--!     (begin <= end implies
--!       Product (begin, end, f) =
--!       APPEND (Product (begin, end - 1, f),
--!         f (end)))"

--! Theta yields f: function from integer
--! to math[Item]
--! parameters
--!   p: integer
--!   "for all n: integer
--!     ((Theta (p)) (n)) =
--!     label (permutation ^ n (p))"

-----
--- Declarations -----
-----

--! type List
--! convention
--!   "0 <= s.length_left and 0 <= s.length_right and
--!   (s.rep.field1 = s.rep.field2 implies
--!     s.length_left = 0) and
--!   (permutation (s.rep.field2) = s.rep.field1 implies
--!     s.length_right = 0) and
--!   permutation^(s.length_left) (s.rep.field1) = s.rep.field2
--!   and
--!   permutation^(s.length_right+1) (s.rep.field2) =
--!     s.rep.field1
--!   and
--!   for all k: integer
--!     ((0 < k and k < s.length_left) implies
--!       permutation^(k) (s.rep.field1) /= s.rep.field2))
--!   and
--!   for all k: integer
--!     ((0 < k and k < s.length_right) implies
--!       permutation^(k+1) (s.rep.field2) /= s.rep.field1))"
--! correspondence
--!   "s.left = Product (1, s.length_left, Theta (s.rep.field1))
--!   and
--!   s.right = Product (1, s.length_right,
--!     Theta (s.rep.field2))"

-----

```

```

--- Local Operations -----
-----

```

```

procedure Swap_Pre_Front (
    r: in out Record2;
    x: in out Position
) renames Swap_Field1;

```

```

procedure Swap_Last_Left (
    r: in out Record2;
    x: in out Position
) renames Swap_Field2;

```

```

--- Exported Operations -----
-----

```

```

procedure Initialize_Package is
begin
    null;
end Initialize_Package;

```

```

procedure Finalize_Package is
begin
    null;
end Finalize_Package;

```

```

procedure Initialize (
    s: in out List
) is
    pre_front: Position;
    last_left: Position;
begin
    Initialize (s.rep);

    Initialize (pre_front);
    Initialize (last_left);

    Copy (pre_front, last_left);
    Swap_Last_Left (s.rep, last_left);
    Swap_Pre_Front (s.rep, pre_front);
--!    s.length_left := 0;
--!    s.length_right := 0;

    Finalize (last_left);
    Finalize (pre_front);
end Initialize;

```

---

```
procedure Finalize (  
    s: in out List  
    ) is  
begin  
    Finalize (s.rep);  
end Finalize;
```

---

```
procedure Swap (  
    s1: in out List;  
    s2: in out List  
    ) is  
begin  
    Swap (s1.rep, s2.rep);  
--!    Swap (s1.length_left, s2.length_left);  
--!    Swap (s1.length_right, s2.length_right);  
end Swap;
```

---

```
procedure Move_To_Start (  
    s: in out List  
    ) is  
    pre_front: Position;  
    last_left: Position;  
begin  
    Initialize (pre_front);  
    Initialize (last_left);  
  
    Swap_Pre_Front (s.rep, pre_front);  
    Copy (pre_front, last_left);  
    Swap_Pre_Front (s.rep, pre_front);  
    Swap_Last_Left (s.rep, last_left);  
--!    s.length_right := s.length_right + s.length_left;  
--!    s.length_left := 0;  
  
    Finalize (last_left);  
    Finalize (pre_front);  
end Move_To_Start;
```

---

```
procedure Move_To_Finish (  
    s: in out List  
    ) is  
    pre_front: Position;  
    last_left: Position;  
begin  
    Initialize (pre_front);  
    Initialize (last_left);
```

```

Swap_Pre_Front (s.rep, pre_front);
Copy (pre_front, last_left);
Swap_Pre_Front (s.rep, pre_front);
Apply_Inverse_Permutation (last_left);
Swap_Last_Left (s.rep, last_left);
--! s.length_left := s.length_right + s.length_left;
--! s.length_right := 0;

```

```

        Finalize (last_left);
        Finalize (pre_front);
end Move_To_Finish;

```

```

-----

procedure Advance (
    s: in out List
) is
    last_left: Position;
begin
    Initialize (last_left);

```

```

        Swap_Last_Left (s.rep, last_left);
        Apply_Permutation (last_left);
        Swap_Last_Left (s.rep, last_left);
--! s.length_left := s.length_left + 1;
--! s.length_right := s.length_right - 1;

```

```

        Finalize (last_left);
end Advance;

```

```

-----

procedure Retreat (
    s: in out List
) is
    last_left: Position;
begin
    Initialize (last_left);

```

```

        Swap_Last_Left (s.rep, last_left);
        Apply_Inverse_Permutation (last_left);
        Swap_Last_Left (s.rep, last_left);
--! s.length_left := s.length_left - 1;
--! s.length_right := s.length_right + 1;

```

```

        Finalize (last_left);
end Retreat;

```

```

-----

procedure Add_Right (
    s: in out List;
    x: in out Item
) is

```

```

        p: Position;
        last_left: Position;
begin
        Initialize (p);
        Initialize (last_left);

        Swap_Label (p, x);
        Swap_Last_Left (s.rep, last_left);
        Transpose_After (last_left, p);
        Swap_Last_Left (s.rep, last_left);
--!      s.length_right := s.length_right + 1;

        Finalize (last_left);
        Finalize (p);
end Add_Right;

-----

procedure Remove_Right (
        s: in out List;
        x: in out Item
    ) is
        p: Position;
        last_left: Position;
begin
        Initialize (p);
        Initialize (last_left);

        Swap_Last_Left (s.rep, last_left);
        Copy (last_left, p);
        Apply_Permutation (p);
        Transpose_After (last_left, p);
        Swap_Last_Left (s.rep, last_left);
        Swap_Label (p, x);
--!      s.length_right := s.length_right - 1;

        Finalize (last_left);
        Finalize (p);
end Remove_Right;

-----

procedure Swap_Rights (
        s1: in out List;
        s2: in out List
    ) is
        p1: Position;
        p2: Position;
begin
        Initialize (p1);
        Initialize (p2);

        Swap_Last_Left (s1.rep, p1);
        Swap_Last_Left (s2.rep, p2);

```

```

Transpose_After (p1, p2);
Swap_Last_Left (s2.rep, p2);
Swap_Last_Left (s1.rep, p1);

Swap_Pre_Front (s1.rep, p1);
Swap_Pre_Front (s2.rep, p2);
Transpose_Before (p1, p2);
Swap_Pre_Front (s2.rep, p2);
Swap_Pre_Front (s1.rep, p1);
--! Swap (s1.length_right, s2.length_right);

Finalize (p2);
Finalize (p1);
end Swap_Rights;

-----

procedure Test_If_At_Start (
    s: in out List;
    at_start: in out Boolean
) is
    pre_front: Position;
    last_left: Position;
begin
    Initialize (pre_front);
    Initialize (last_left);

    Swap_Pre_Front (s.rep, pre_front);
    Swap_Last_Left (s.rep, last_left);
    Test_If_Equal (pre_front, last_left, at_start);
    Swap_Last_Left (s.rep, last_left);
    Swap_Pre_Front (s.rep, pre_front);

    Finalize (last_left);
    Finalize (pre_front);
end Test_If_At_Start;

-----

procedure Test_If_At_Finish (
    s: in out List;
    at_finish: in out Boolean
) is
    pre_front: Position;
    last_left: Position;
    p: Position;
begin
    Initialize (pre_front);
    Initialize (last_left);
    Initialize (p);

    Swap_Pre_Front (s.rep, pre_front);
    Swap_Last_Left (s.rep, last_left);
    Copy (last_left, p);

```

```

    Apply_Permutation (p);
    Test_If_Equal (pre_front, p, at_finish);
    Swap_Last_Left (s.rep, last_left);
    Swap_Pre_Front (s.rep, pre_front);

    Finalize (p);
    Finalize (last_left);
    Finalize (pre_front);
end Test_If_At_Finish;

end Two_Way_List_Template;

```

#### 4.4.3 Implementing Permutation\_Template

The precondition for `One_Way_Nilpotent_Template`'s `Change_Target` operation prohibits the creation of circular linked structures. This requirement permits operations that work in constant time and also allows the use of reference counting. If we eliminate `Change_Target`'s requirement, allowing circular structures, then we lose this efficiency (see Section 4.2.1). It seems that `Permutation_Template` should have the same problem, and indeed it would if not for a clever implementation trick.

The implementation of `Permutation_Template` might use a circular doubly linked list. For example, suppose a client program instantiated `Permutation_Template` with `Character` and created a cyclic linked structure having three positions with the labels "a," "x," and "r." Conceptually, this can be illustrated as follows:

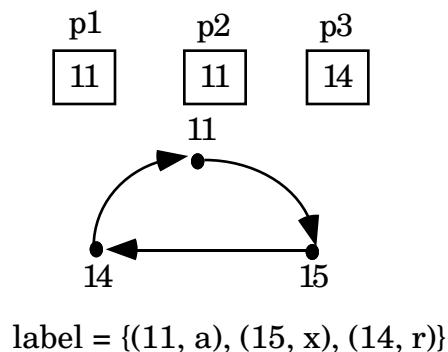


Figure 32 — Cyclic Linked Structure With Three Positions

A circular doubly linked list implementation might look like the following:

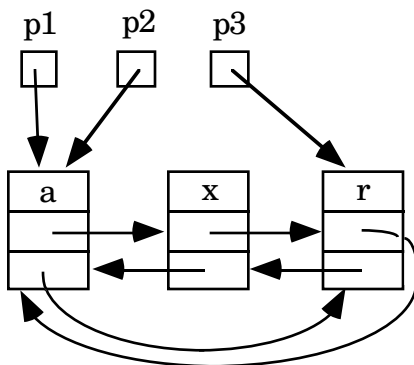


Figure 33 — Circular Doubly Linked List Implementation

One might try to base the reference counts for each node on the number of successor/predecessor references plus the number of Position variable references, i.e., in the above diagram, the node with label “a” would have a reference count of four. But this does not work because when all the Position variable are finalized, each node has a reference count of two. When Finalize finalizes the last Position variable it sees that the node referenced has a reference count greater than zero and should not reclaim the node (or any of the other nodes in the list). [Meyer 88, page 365] constructs an example similar to this and declares “reference counting is not a general enough technique: it fails to recycle cyclic structures.” But this claim turns out to be false.

The reference count needs to be based on the number of Position variables that reference the linked structure. When the last Position variable is finalized, the reference count goes to zero, and Finalize can then reclaim the entire linked structure. But the problem is where to store this reference count. One might try to store the reference counts in the nodes, i.e., the count stored in the node equals the number of Position variables referencing that node. Determining if the linked structure’s count equals zero would then require traversing the entire structure to examine each node’s count. Traversing the linked structure leads to performance linear in the length of the structure. An alternative to this approach must be found because linear performance is unacceptable; constant time is the goal.

An alternative is to maintain a second “shadow” list whose total number of nodes always equals the number of different Position variable values in the original cycle. When there is only one Position variable left, there is only one node in the shadow list. When the last Position variable is finalized, Finalize can determine that there is only one node in the shadow list, and then reclaim storage. The following illustrates this representation:

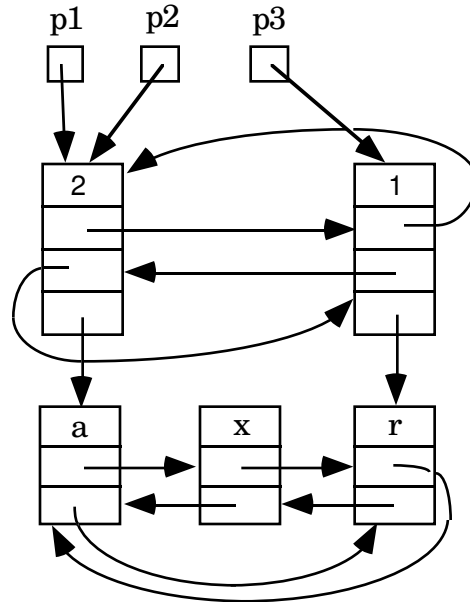


Figure 34 — Representation Using a Shadow List

Each of `Permutation_Template`'s operations must maintain both the original list (which holds the data) and the shadow list. An implementation for `Permutation_Template` can be found in Appendix B. It maintains the shadow list, but does not achieve constant time finalization. Constant time finalization can be achieved using a technique similar to that described for `One_Way_Nilpotent_Template` (see Section 4.3.3).

#### 4.5 Abstract Module State in `One_Way_Nilpotent_Template` and `Permutation_Template`

Both `One_Way_Nilpotent_Template` and `Permutation_Template` maintain abstract module state. This violates Principle 2 of Section 2.1.2. Does this mean that we lose local certification of correctness because of intermodule interference (see Section 2.4)? This section address that question.

The following describes how intermodule interference might arise because of component-level abstract state. Suppose a client program “with”s and instantiates `One_Way_Nilpotent_Template`. It also “with”s and instantiates `Queue_16`, passing it the instance of `One_Way_Nilpotent_Template`. Then the client program might be able to interfere with the correct operation of `Queue_16` because they both share the same instance of `One_Way_Nilpotent_Template` and the client might change this instance's abstract state in such a way as to violate a convention assumed by `Queue_16`'s package body or vice versa.

Hypothetically, how might the client program do this?

- It might declare, initialize, and label its own `Position` variable that happens to have the same value as one of `Queue_16`'s. But this is not possible, since `Position` values are dispensed one at a time by `Label_New_Position`, based on the value of the

One\_Way\_Nilpotent variable “used.” Each time Label\_New\_Position is called, “used” is incremented by one, so a newly labeled Position variable cannot have the same value as an existing Position variable.

- The client program might use Copy to copy a Position used in Queue\_16, or vice versa. However, Queue\_16 is not designed to pass Position variables back and forth, so this is not possible either. Even if it were, the copy would show up in the specification and reasoning could still be done locally.
- Some other One\_Way\_Nilpotent\_Template operation might allow the client program to access a part of the label or target functions that are reachable from Queue\_16 Positions. For example, Queue\_16 links Positions together when an Item is enqueued. All these linked-together Positions are reachable from Queue\_16 by applying the target function to a Position maintained by Queue\_16. It is not possible for the client program to access a part of the label or target functions that are reachable from Queue\_16 Positions because all One\_Way\_Nilpotent\_Template operations that change the label or target functions, only change them for the Positions that they are passed. Since the client program does not have access to Queue\_16’s Positions, it cannot use One\_Way\_Nilpotent\_Template operations to access the parts of label or target that are reachable from Queue\_16 Positions.

Because of its design, there is no way for two clients of the same instance of One\_Way\_Nilpotent\_Template to interfere with each other. The above arguments apply to Permutation\_Template as well.

This leads us to the conclusion that Principle 2 might be too restrictive. Indeed it is in theory, but it is not so easy to design components that maintain abstract module state while at the same time not permitting intermodule interference. Therefore we recommend strict observation of Principle 2, and allow these two exceptions which bootstrap from raw Ada to provide us support for constructing unbounded linked structures. Other exceptions (e.g., for I/O) may also be useful but should be treated carefully.

## 4.6 Summary

This chapter discussed two topics: the additional machinery required of the built-in scalars when used with RESOLVE/Ada components; and how to encapsulate the type constructors so that they can be safely used with RESOLVE/Ada components. Bootstrapping from raw Ada refers to the fact that the encapsulation of Ada’s type constructors produces components whose package specification conforms to the principles of Section 2.1, but whose package implementations are implemented in “raw” Ada, i.e., not using RESOLVE/Ada components and not necessarily conforming to the principles of Sections 2.2 and 2.3.

Section 4.1 introduced the Built\_In\_Types package which eliminates the need to create separate components for providing the types Boolean, Character, Integer, and Float. This package permits the component implementer to use these built-in scalar types in conjunction with RESOLVE/Ada components. The Built\_In\_Types package is “with”ed and “use”d by client programmers or component implementers whenever one of Ada’s Boolean, Character, Integer, or Float scalar types is needed. It provides the following operations for each of these types: Initialize, Finalize, Swap, Copy, Test\_If\_Equal and Display.

Section 4.2.1 discussed the encapsulation of Ada's record type constructor. Encapsulating Ada records so they can be readily used with RESOLVE/Ada components requires building an interface that conforms to the interface principles of Section 2.1. Implications of this are:

- A record is now provided by a generic package, i.e., a record with two fields is provided by `Record2_Template`, a record with three fields is provided by `Record3_Template`, etc.
- A client must “with” and instantiate a `Record_Template` component in order to use a record in the client program.
- A record now has `Initialize`, `Finalize` and `Swap` operations.
- Data is moved in and out of the record by swapping, not by assignment.

Section 4.2.2 discussed the encapsulation of Ada's array type constructor. The implications listed above for `Record_Template` components apply to `Array_Template` components as well. Additionally, since the basic mathematical model used for an array is a function from Integer to Item, many implementation variations are possible, including ones that result in constant-time performance for every operation. This section introduced the `Static_Array_Template`, so called because the upper and lower bounds are fixed at instantiation time. Another variation might allow setting different bounds for each array variable after it is declared.

Section 4.3 discussed the encapsulation of Ada's access type constructor. Two components were introduced, `One_Way_Nilpotent_Template` for building acyclic linked structures and `Permutation_Template` for building cyclic linked structures. Both of these components can be implemented so that their operations execute in constant time and so that they reclaim storage automatically. An implication of automatic storage reclamation is that no storage leaks or dangling references can be created, so long as a client program adheres to the client implementation principles of Section 2.2. Furthermore, with these two components, the client programmer can build and reason about linked structures based on the provided abstractions instead of building and reasoning based on concrete things like pointers and nodes. Finally, we noted that both `One_Way_Nilpotent_Template` and `Permutation_Template` maintain abstract module-level state, which seems to jeopardize our chances for local certification of correctness. Fortunately, both are designed so that no intermodule interference can occur between clients, and therefore we maintain local certification of correctness.