Ada.Characters

Introduction

Character Capabilities

- Package `Ada.Characters` is the parent package for identification and manipulation of characters
  - `Ada.Characters.Handling` - queries and conversion subprograms
  - `Ada.Characters.Latin_1` - constants for character values 0 .. 255

Ada.Characters.Latin_1

Package Contents (Partial)

```
package Ada.Characters.Latin_1 is
   NUL             : constant Character := Character'Val (0);
   SOH             : constant Character := Character'Val (1);
   STX             : constant Character := Character'Val (2);
   ETX             : constant Character := Character'Val (3);
   EOT             : constant Character := Character'Val (4);
   ENQ             : constant Character := Character'Val (5);
   -- ...
   Space           : constant Character := ' ';  -- Character'Val(32)
   Exclamation     : constant Character := '!';  -- Character'Val(33)
   Quotation       : constant Character := '"';  -- Character'Val(34)
   Number_Sign     : constant Character := '#';  -- Character'Val(35)
   Dollar_Sign     : constant Character := '$';  -- Character'Val(36)
   -- ...
   LC_A            : constant Character := 'a';  -- Character'Val (97)
   LC_B            : constant Character := 'b';  -- Character'Val (98)
   LC_C            : constant Character := 'c';  -- Character'Val (99)
   LC_D            : constant Character := 'd';  -- Character'Val (100)
   LC_E            : constant Character := 'e';  -- Character'Val (101)
   -- ...
end Ada.Characters.Latin_1;
```

Idioms

- Obvious - giving names to unprintable characters
- Good coding practice to use names instead of literals
  - Easier searching for non-alphanumeric characters
- Some symbols have multiple names, such as:
  - `Minus_Sign` → `Hyphen`
  - `NBSP` → `No_Break_Space`
  - `Ring_Above` → `Degree_Sign`

Ada.Characters.Handling

Character Queries

- Boolean functions whose return is based on the *category* of the character, such as:

```
function Is_Control           (Item : Character) return Boolean;
function Is_Graphic           (Item : Character) return Boolean;
function Is_Letter            (Item : Character) return Boolean;
function Is_Lower             (Item : Character) return Boolean;
function Is_Upper             (Item : Character) return Boolean;
function Is_Basic             (Item : Character) return Boolean;
function Is_Digit             (Item : Character) return Boolean;
function Is_Decimal_Digit     (Item : Character) return Boolean
  renames Is_Digit;
function Is_Hexadecimal_Digit (Item : Character) return Boolean;
function Is_Alphanumeric      (Item : Character) return Boolean;
```

Character Transformation

- Functions to force case

```
function To_Lower (Item : in Character) return Character;
function To_Upper (Item : in Character) return Character;
```

- Functions to force case (string version)

```
function To_Lower (Item : in String) return String;
function To_Upper (Item : in String) return String;
```

- Functions to convert to/from `Wide_Character` and `Wide_String`

```
function To_Character (Item       : Wide_Character;
                       Substitute : Character := ' ')
                       return Character;
function To_String (Item       : Wide_String;
                    Substitute : Character := ' ')
                    return String;
function To_Wide_Character (Item : Character)
                            return Wide_Character;
function To_Wide_String (Item : String)
                         return Wide_String;
```

Lab

Ada.Characters Lab

- Requirements
  - Read an integer value (representing ASCII) from the console
  - Convert the integer value to its character equivalent
  - Print a result according to the following rules:
    - If the character is a letter, convert it to the opposite case
    - If the character is not a letter but it is printable, print it
    - If the character is a line terminator, print its name
    - If none of the above apply, just print out "unprintable"

Ada.Characters Lab Solution

```
with Ada.Text_IO;              use Ada.Text_IO;
with Ada.Characters.Handling; use Ada.Characters.Handling;
with Ada.Characters.Latin_1;  use Ada.Characters.Latin_1;
procedure Main is
   Input : String (1 .. 5);
   Last  : Natural;

   procedure Print (Char : Character) is
   begin
      Put ("Result: ");
      if Is_Letter (Char) then
         if Is_Upper (Char) then
            Put (To_Lower (Char));
         else
            Put (To_Upper (Char));
         end if;
      elsif Is_Graphic (Char) then
         Put (Char);
      elsif Is_Line_Terminator (Char) then
         case Char is
            when LF  => Put ("LF");
            when VT  => Put ("VT");
            when FF  => Put ("FF");
            when CR  => Put ("CR");
            when NEL => Put ("NEL");
            when others =>
               Put ("Missed one: " & Integer'image (Character'pos (Char)));
         end case;
      else
         Put ("unprintable");
      end if;
      New_Line;
   end Print;

begin
   Put ("Enter ASCII value: ");
   Get_Line (Input, Last);
   Print (Character'val (Integer'value (Input (1 .. Last))));
end Main;
```

Summary

Summary

- `Ada.Characters` contains consistent mechanisms for
  - Referring to unprintable and special characters
  - Queries on the properties of characters
- Same capabilities for other character sets in `Ada.Wide_Characters` and `Ada.Wide_Wide_Characters`

Ada.Strings

Introduction

Predefined Type `String`

- `String` type allows varying lengths, but `String` objects are fixed lengths
  - It's just an unconstrained array of characters
- Language does not have any built-in string manipulation subprograms
- What if we want to change the length of the object?

Ada.Strings.Fixed

- Based on fixed-length string
- Strings are unconstrained arrays, so objects cannot change length
- Operations that return string of unknown (or different) length can only be used for initialization

Ada.Strings.Bounded

- Contains generic package
  - Must create instance passing in maximum string length
- String length is maintained internally
  - Operations can modify objects in-place
  - Subject to limit of maximum length
- Contains query to get maximum length
  - Allows client to pre-determine if length will be exceeded

Ada.Strings.Unbounded

- Not a generic package
  - No maximum length (except run-time limits!)
- String length is maintained internally
  - Operations can modify objects in-place
  - Subject to limit of maximum length
- Requires dynamic memory allocation

String Operations

Primitive String Functions

- Operations like concatenation ("&") and comparison (">=", etc)
  - Built in for **fixed-length** strings
  - Defined in appropriate package for **bounded** and **unbounded**
    - Require `use` or `use type` for simple visibility

Common Subprograms

`"*"`	Return the character or string duplicated N times
`Count`	Number of occurrences of specified string/character set
`Delete`	Remove slice
`Find_Token`	Location of token that matches/doesn't match character set
`Head`	Front N characters (padded as necessary)
`Index`	Index of character/string, given starting location/direction
`Index_Non_Blank`	Index of first/last character/string, given starting location/direction
`Insert`	Insert substring into source before the specified position
`Overwrite`	Overwrite source with new substring starting at the specified position
`Replace_Slice`	Replace specified slice with new string
`Tail`	Last N characters (padded as necessary)
`Translate`	Translate string using specified character mapping
`Trim`	Remove leading/trailing characters from source

Bounded/Unbounded Subprograms

`Append`	Concatenate bounded strings and/or standard strings to create an unbounded string
`Element`	Character at specified position
`Length`	Length of string
`Replace_Element`	Put input character specified position
`Slice`	Standard string slice from specified positions
`To_String`	Convert unbounded string to standard string

Unique Subprograms

- `Ada.Strings.Fixed`

`Move`	Copy source to target with truncation/padding

- `Ada.Strings.Bounded`

`Bounded_Slice`	Bounded string slice from specified positions
`Replicate`	Return the bounded string duplicated N times
`Set_Bounded_String`	Procedural copy standard string to bounded string
`To_Bounded_String`	Copy standard string to bounded string

- `Ada.Strings.Unbounded`

`Set_Unbounded_String`	Procedural copy standard string to unbounded string
`To_Unbounded_String`	Copy standard string to unbounded string
`Unbounded_Slice`	Unbounded string slice from specified positions

Lab

Ada.Strings Lab

- Requirements
  - Create a (simplistic) source code parser to read an Ada file
    - Use your main program as input!
  - Print the number of comments and semi-colons in the file
  - Print a sorted list of objects found in the code
- Hints
  - Object name will be identifier before a standalone ":"
  - Object list will need varying length strings
- Extra Credit (if you have time):
  - When you search for strings, you will also find them as a search parameter!
    - Find a way to "skip over" string literals

Ada.Strings Lab Solution (Declarations)

```
with Ada.Text_IO; use Ada.Text_IO;
with Ada.Strings.Fixed;
with Ada.Strings.Unbounded;
procedure Main is

   -- hard-coded filename
   Filename : constant String := "main.adb";

   File : File_Type;
   Line : String (1 .. 100);
   Last : Natural;

   Objects      : array (1 .. 100) of Ada.Strings.Unbounded.Unbounded_String;
   Object_Count : Natural := 0;

   Comments   : Natural := 0;
   Semicolons : Natural := 0;

   Colon : Natural;
```

Ada.Strings Lab Solution (Main)

```
begin

   Open (File, In_File, Filename);
   while not End_Of_File (File) loop
      Get_Line (File, Line, Last);
      declare
         Stripped_Line : constant String := Hide_Strings (Line (1 .. Last));
      begin
         Comments := Comments + Ada.Strings.Fixed.Count
             (Source  => Stripped_Line,
              Pattern => "--");
         Semicolons := Semicolons + Ada.Strings.Fixed.Count
             (Source  => Stripped_Line,
              Pattern => ";");

         Colon := Ada.Strings.Fixed.Index
             (Source  => Stripped_Line,
              Pattern => " : ");
         if Colon in Stripped_Line'range then
            Object_Count           := Object_Count + 1;
            Objects (Object_Count) :=
              Ada.Strings.Unbounded.To_Unbounded_String
                (Stripped_Line (1 .. Colon));
            Ada.Strings.Unbounded.Trim
              (Objects (Object_Count), Ada.Strings.Both);
         end if;
      end;
   end loop;
   Close (File);

   Put_Line ("Comments: " & Integer'image (Comments));
   Put_Line ("Semi-colons: " & Integer'image (Semicolons));

   declare
      Hold : Ada.Strings.Unbounded.Unbounded_String;
      use type Ada.Strings.Unbounded.Unbounded_String;
   begin
      for I in 1 .. Object_Count loop
         for J in 1 .. Object_Count - 1 loop
            if Objects (J) > Objects (J + 1) then
               Hold            := Objects (J);
               Objects (J)     := Objects (J + 1);
               Objects (J + 1) := Hold;
            end if;
         end loop;
      end loop;
   end;

   Put_Line ("Objects: ");
   for I in 1 .. Object_Count loop
      Put_Line ("  " & Ada.Strings.Unbounded.To_String (Objects (I)));
   end loop;
end Main;
```

Ada.Strings Lab Solution (Extra Credit)

```
function Hide_Strings
  (Str : String)
   return String is
   First : Natural;
   Last  : Natural;
begin
   First := Ada.Strings.Fixed.Index (Str, """");
   if First in Str'range then
      Last := Ada.Strings.Fixed.Index
          (Source  => Str (First + 1 .. Str'last),
           Pattern => """");
      if Last in Str'range then
         return Ada.Strings.Fixed.Replace_Slice
            (Str, First, Last, "");
      end if;
   end if;
   return Str;
end Hide_Strings;
```

Summary

Summary

- `Ada.Strings.Fixed`
  - String operations for **String**
- `Ada.Strings.Bounded`
  - Varying length string where the maximum length is constrained
  - Requires generic instantiation
  - Implementation may be handled without dynamic memory allocation
- `Ada.Strings.Unbounded`
  - Varying length string with no maximum length
  - Implementation typically requires dynamic memory allocation

Ada.Text_IO

Introduction

Ada.Text_IO

- Most common I/O library unit - works with normal text I/O
- Works with **string** types
  - **Ada.Wide_Text_IO** for **wide_string**
  - **Ada.Wide_Wide_Text_IO** for **wide_wide_string**
- Other I/O packages (not discussed in this module):
  - **Ada.Sequential_IO** and **Ada.Direct_IO**
    - Operations on binary files for elements of a given type
  - **Ada.Storage_IO**
    - Operations on reading/writing to/from memory buffer
  - **Ada.Streams.Stream_IO**
    - Operations for streaming data to/from binary files

```
declare
   -- read from default input file
   From_Input : constant String := Ada.Text_IO.Get_Line;
begin
   -- write to default output file
   Ada.Text_IO.Put_Line ("I just typed: " & From_Input);
end;
```

Scalar Type I/O

- Child generic packages of **Ada.Text_IO** to read / write scalar types
  - **Ada.Text_IO.Integer_IO**
  - **Ada.Text_IO.Modular_IO**
  - **Ada.Text_IO.Float_IO**
  - **Ada.Text_IO.Fixed_IO**
  - **Ada.Text_IO.Decimal_IO**
  - **Ada.Text_IO.Enumeration_IO**
- Create instances of the generic package to read/write

```
declare
   type Float_T is digits 6;
   package Float_IO is new Ada.Text_IO.Float_IO (Float_T);
   F : Float_T;
begin
   -- Read floating point number from default input file
   Float_IO.Get (F);
   -- Writing floating point number to default output file
   Float_IO.Put (F * 10.0, Fore => 1, Aft => 2, Exp => 3);
end;
```

File Input/Output

Standard Input / Output

- **Ada.Text_IO** maintains *default* input and output files

```
-- reads from default input file
S : constant string := Get_Line;
-- ...
-- writes to default output file
Put_Line ( S );
```

- At initialization, default input and output refer to the console
  - Which is why all our previous usage was so simple!

Files

- Files can be created (new for writing) or opened (for reading, writing, or appending)
  - File modes:
    - **In_File** → Open for reading
    - **Out_File** → Reset file and open for writing
    - **Append_File** → Position file at end and open for writing

```
declare
   File : File_Type;
begin
   Create (File => File,
           Mode => Out_File,
           Name => "foo.txt");
   Put_Line (File, "Line 1");
   Close (File);
   -- This "Open" is only legal because "foo.txt" already exists
   Open (File, Out_File, "foo.txt");
   Put_Line (File, "Line 2");
   Close (File);
   Open (File, Append_File, "foo.txt");
   Put_Line (File, "Line 3");
   Close (File);
   Open (File, In_File, "foo.txt");
   -- Read lines from file and print to standard output
   Put_Line (Get_Line (File));
   Put_Line (Get_Line (File));
end;
```

File Status Queries

End_Of_File	Check if end of file has been reached
Is_Open	Check if file has been opened (regardless of file mode)
Mode	Return how file was opened
Name	Name of open file
Col	Current column in file
Line	Current line in file

Type-Specific I/O

Ada.Text_IO.Integer_IO

```
declare
   type Integer_T is range -1_000 .. 1_000;
   package Io is new Ada.Text_IO.Integer_IO (Integer_T);
   I : Integer_T;
begin
   Io.Get (I);
   Io.Put
     (Item  => I,
      Width => 10,  -- optional: minimum number of characters to print
      Base  => 16); -- optional: numeric base
end;
```

- **Get** will read until a non-numeric character is encountered, ignoring leading or trailing whitespace
  - **123** will set I to 123
  - **45X67** will set I to 45
- `IO` has global objects `Default_Width` and `Default_Base` which can be modified to set default values for like-named parameters
- `Ada.Text_IO.Modular_IO` behaves the same

Ada.Text_IO.Float_IO

```
declare
   type Float_T is digits 6 range -100.0 .. 100.0;
   package Io is new Ada.Text_IO.Float_IO (Float_T);
   F : Float_T;
begin
   Io.Get (F);
   Io.Put
     (Item => F,
      Fore => 1,    -- optional: number of digits before decimal point
      Aft  => 2,    -- optional: number of digits after decimal point
      Exp  => 3);   -- optional: numeric of characters for exponent
end;
```

- **Get** will read until a non-numeric character is encountered, ignoring leading or trailing whitespace
  - **12** will set F to 12.0
  - **23.45.67** will set F to 23.45
- `IO` has global objects `Default_Fore`, `Default_Aft` and `Default_Exp` which can be modified to set default values for like-named parameters
- **Ada.Text_IO.Fixed_IO** and **Ada.Text_IO.Decimal_IO** behave the same

Ada.Text_IO.Enumeration_IO

```
declare
   type Enumeration_T is ( Red, Yellow, Green );
   package Io is new Ada.Text_IO.Enumeration_IO (Enumeration_T);
   E : Enumeration_T;
begin
   Io.Get (E);
   Io.Put
     (Item => F,
      Width => 10,           -- optional: minimum number of characters to print
      Set   => Lower_Case); -- optional: flag for Upper_Case or Lower_Case
end;
```

- **Get** will read until the end of the line or trailing whitespace, case-insensitive
  - **YelloW** will set **E** to **Yellow**
  - **Red Blue** will set **E** to **Red**
- `IO` has global objects `Default_Width` and `Default_Setting` which can be modified to set default values for like-named parameters

Exceptions

Ada.IO_Exceptions

- I/O Packages have common exceptions (defined in **Ada.IO_Exceptions** and renamed in **Ada.Text_IO** for easier reference)
- The most common Text I/O exceptions:
  - `Status_Error` → Raised on **Open**/**Create** if file being opened/created is already open. For any other operation, raised if file is not open
  - `Name_Error` → Raised if filename is invalid for **Open**/**Create**
  - `Use_Error` → Raised if unable to **Open**/**Create**
  - `Data_Error` → Failure of **Get** to read valid data

Lab

Ada.Text_IO Lab

- Requirements
  - Create an enumerated type
  - Use the console to query the user how many inputs (N) will follow
  - Use the console to query the user N times for an enumeral
  - If the enumeral is valid, write the index and enumeral to a file
    - Else write an error message to the console
  - When all inputs were read, echo the file to the console
- Hints
  - Use instantiations of the type-specific I/O packages to handle console queries
    - Better error handling
  - Use Text_IO to echo the file to the console

Ada.Text_IO Lab Solution

```
with Ada.Text_IO; use Ada.Text_IO;
procedure Main is
   type Enumerated_T is (Red, Yellow, Green);
   package Enum_Io is new Enumeration_IO (Enumerated_T);
   type Count_T is mod 10;
   package Count_Io is new Modular_Io (Count_T);
   E    : Enumerated_T;
   C    : Count_T;
   File : File_Type;
begin

   Put ("Count: ");
   Count_Io.Get (C);

   Create (File, Out_File, "foo.txt");

   for I in 1 .. C loop
      Count_Io.Put (I, Width => 3);
      Count_Io.Put (File, I, Width => 3);
      Put (" => ");
      begin
         Enum_Io.Get (E);
         Enum_Io.Put (File, E, Width => 10);
      exception
         when others =>
            Put_Line ("Something didn't look right");
      end;
      New_Line (File);
   end loop;
   Close (File);

   Put_Line ("Echoing file");
   Open (File, In_File, "foo.txt");
   while not End_Of_File (File) loop
      Put_Line (Get_Line (File));
   end loop;
end Main;
```

Summary

Summary

- `Ada.Text_IO` is the most common text input/output processing process
- Text_IO has simple mechanisms to read scalar types
  - 'Image and 'Value work, but are simplistic
    - 'Image does not allow formatting of output
    - 'Value will fail if entire input cannot be converted

Containers

Introduction

Container Library

- `Ada.Containers` parent package
- Packages (including generics)
  - Different types of data containers
  - Hold an `Element` type
  - Container types are `tagged`
- Types defined as a product of both
  - A data structure
  - An implementation
  - Define some added operations
- Containers share sets of operations
  - Seen later

Container Types

Data Structures (1/2)

- Vector
  - Essentially an array
  - *Capacity* and size can differ
- Doubly-linked list
  - Linked list
  - Iteration in both directions
- Map
  - Containers matching Key -> Element
  - Not a one-to-one relationship
    - Can have several keys for a single element
- Set
  - Collection of **unique** values
- Queue
  - No iterator
  - Only ordered access
  - For multi-tasking operations

Data Structures (2/2)

Ada 2012

- Tree
  - Similar to list
  - A node can have several children
- Holder
  - Wraps around an indefinite (unconstrained, classwide, incomplete...)
  - Resulting type is definite
  - Single element, no iteration or cursor

Implementations (1/2)

- *Bounded*
  - Maximal storage is bounded
  - Constant capacity and element size
  - Only static allocation
  - `Bounded_<Structure>`
- *Unbounded*
  - Capacity can grow dynamically
  - Easiest to use
  - Default
- *Ordered*
  - Elements are sorted in order
  - Must provide < and = operators
  - Not hashed
  - `XXX_Ordered_<Structure>`
- *Hashed*
  - Elements are hashed
  - Must provide `Hash` function and = operator
  - Not ordered
  - Some hash functions are provided (e.g. `Ada.Strings.Hash`)
  - `XXX_Hashed_<Structure>`

Implementations (2/2)

Ada 2012

- *Indefinite*
  - Element can be indefinite
  - Size of element is unknown
  - `Indefinite_XXX_<Structure>`

Example of Containers

- Standard defines 25 different container variations
- `Indefinite_Vector`
  - Static capacity
  - Dynamically sized (indefinite elements)
  - Random access in `O(1)`
- `Ordered_Set`
  - Unique elements
  - Differenciated by < and =
  - Manipulated in order
- `Bounded_Doubly_Linked_List`
  - Static size of container and elements
  - Insertions and deletions in `O(1)`

Declaration

- Generic packages
- Always need at least the `Element_Type`
- Examples chosen for the next slides:

```
package Pkg_Vectors is new Ada.Containers.Bounded_Vectors
   (Index_Type => Index_Type, Element_Type => Integer
-- "=" (A, B : Integer) is directly visible
);
```

```
package Pkg_Sets is new Ada.Containers
   .Indefinite_Ordered_Sets
   (Element_Type => String);
```

```
package Pkg_Maps is new Ada.Containers.Hashed_Maps
   (Key_Type => Ada.Strings.Unbounded.Unbounded_String,
    Element_Type    => Float,
    Hash            => Ada.Strings.Unbounded.Hash,
    Equivalent_Keys => Ada.Strings.Unbounded."=");
```

Instanciation

- May require an initial `Empty_xxx` value

```
Student_Per_Day : Pkg_Vectors.Vector (5);
-- Warning: initial size is 0, using an Empty_Vector as
--          initial value would mean a *capacity* of 0!

Received_Parcels : Pkg_Sets.Set := Pkg_Sets.Empty_Set;

Math_Constants : Pkg_Maps.Map := Pkg_Maps.Empty_Map;
```

Containers Operations

Common Operations

- Lots of common operations
  - What is available depends greatly on the exact container type
  - ... so does syntax
- Insertion
- Iteration
- Comparison
- Sort
- Search

Insertion

- May be in order `Append` or `Prepend`
- May be `Insert` (at random or at given index)
- May `Replace` an existing element

```
Student_Per_Day.Append (10);
Student_Per_Day.Append (8);
Student_Per_Day.Append (9);

Received_Parcels.Insert ("FEDEX AX431661VD");
Received_Parcels.Insert ("UPS ZZ-44-I12");

Math_Constants.Insert
  (To_Unbounded_String ("Pi"), 3.141_59);
Math_Constants.Insert (To_Unbounded_String ("e"), 2.718);
```

Iteration

- Container have a `Cursor` type
  - Points to an element in a container
  - Can be used for advanced iterations

```
for Student_Count of Student_Per_Day loop
   Put_Line (Integer'Image (Student_Count));
end loop;

for Parcel_Id of Received_Parcels loop
   Put_Line (Parcel_Id);
end loop;

-- We use the cursor to have both key and value
for C in Math_Constants.Iterate loop
   Put_Line
     (To_String (Key (C)) & " = " &
      Float'Image (Element (C)));
end loop;
```

Comparison

```
-- xxx2 are objects with the exact same content
pragma Assert (Student_Per_Day = Student_Per_Day2);
pragma Assert (Received_Parcels = Received_Parcels2);
pragma Assert (Math_Constants = Math_Constants2);

-- After changing the content, equality does not hold
Student_Per_Day.Append (10);
Received_Parcels.Insert ("Chronopost 13214GUU-035");
Math_Constants.Insert (To_Unbounded_String ("G"), 9.8);

pragma Assert (Student_Per_Day /= Student_Per_Day2);
pragma Assert (Received_Parcels /= Received_Parcels2);
pragma Assert (Math_Constants /= Math_Constants2);
```

Sort

- Arrays
  - **Ada.Containers.Generic_Array_Sort**
  - **Ada.Containers.Generic_Constrained_Array_Sort**
- Any object that has indexing
  - **Ada.Containers.Generic_Sort**

```
procedure Sort
  (V    : in out Pkg_Vectors.Vector; First : Index_Type;
   Last :        Index_Type)
is
   procedure Swap_Object (A, B : Index_Type) is
      Temp : Integer := V (A);
   begin
      V (A) := V (B);
      V (B) := Temp;
   end Swap_Object;

   procedure Sort_Object is new Ada.Containers
     .Generic_Sort
     (Index_Type => Index_Type, Before => "<",
      Swap       => Swap_Object);
begin
   Sort_Object (First, Last);
end Sort;
```

Search

- Use Find for a Cursor
  - <Pkg>.No_Element is a Cursor if not found
- Use Find_Index for an Index_Type (vectors)

```
C : constant Pkg_Vectors.Cursor :=
  Student_Per_Day.Find (10);
C2 : constant Pkg_Sets.Cursor :=
  Received_Parcels.Find ("UPS ZZ-44-I12");
C3 : constant Pkg_Maps.Cursor :=
  Math_Constants.Find
    (To_Unbounded_String
       ("Pi")); -- Finds by the key!
```

Reference

Standard Ada.Containers Packages

- Definite Types
  - Vectors
  - Doubly_Linked_Lists
  - Multiway_Trees
  - Hashed_Maps
  - Ordered_Maps
  - Hashed_Sets
  - Ordered_Sets
- Indefinite Types
  - Indefinite_Vectors
  - Indefinite_Doubly_Linked_Lists
  - Indefinite_Multiway_Trees
  - Indefinite_Hashed_Maps
  - Indefinite_Ordered_Maps
  - Indefinite_Hashed_Sets
  - Indefinite_Ordered_Sets
  - Indefinite_Holders
- Bounded Types
  - Bounded_Vectors
  - Bounded_Doubly_Linked_Lists
  - Bounded_Multiway_Trees
  - Bounded_Hashed_Maps
  - Bounded_Ordered_Maps
  - Bounded_Hashed_Sets
  - Bounded_Ordered_Sets

Lab

Containers Lab

- Requirements
  - Create a database of various information about various cities
  - Populate the database
    - No requirement to add all information for each city at the same time
  - Print the database
    - For extra credit: Cities / information should be sorted
- Hints
  - Use a **map** ADT to organize data by city
  - Multliple methods to organize city information
    - Array, list, vector, etc

Containers Lab Solution - Database (Spec)

```
with Ada.Containers.Bounded_Vectors;
with Ada.Strings.Unbounded; use Ada.Strings.Unbounded;
package City_Trivia is

   package Strings_Vector is new Ada.Containers.Bounded_Vectors
    (Index_Type => Natural, Element_Type => Unbounded_String);
   subtype Strings_Vector_T is Strings_Vector.Vector (100);

   procedure Add_Trivia
    (City        : String;
     Information : String);

   function Get_Trivia
    (City : String)
     return Strings_Vector_T;
   function Get_Keys return Strings_Vector_T;

   package Sort is new Strings_Vector.Generic_Sorting;

end City_Trivia;
```

Containers Lab Solution - Database (Body)

```
with Ada.Containers.Bounded_Ordered_Maps;
package body City_Trivia is
  use type Strings_Vector_T;
  package Maps is new Ada.Containers.Bounded_Ordered_Maps
   (Key_Type => Unbounded_String, Element_Type => Strings_Vector_T);
  use type Maps.Cursor;
  Map : Maps.Map (100);

  procedure Add_Trivia (City        : String;
                        Information : String) is
    Key    : constant Unbounded_String := To_Unbounded_String (City);
    Info   : constant Unbounded_String := To_Unbounded_String (Information);
    Cursor : Maps.Cursor;
    Vector : Strings_Vector_T;
  begin
    Cursor := Map.Find (Key);
    if Cursor = Maps.No_Element then
        Vector.Append (Info);
        Map.Insert (Key      => Key,
                    New_Item => Vector);
    else
        Vector := Maps.Element (Cursor);
        Vector.Append (Info);
        Map.Replace_Element (Position => Cursor,
                             New_Item => Vector);
    end if;
  end Add_Trivia;

  function Get_Trivia (City : String) return Strings_Vector_T is
    Ret_Val : Strings_Vector_T;
    Key     : constant Unbounded_String := To_Unbounded_String (City);
    Cursor  : Maps.Cursor;
  begin
    Cursor := Map.Find (Key);
    if Cursor /= Maps.No_Element then
      Ret_Val := Maps.Element (Cursor);
    end if;
    Sort.Sort (Ret_Val);
    return Ret_Val;
  end Get_Trivia;

  function Get_Keys return Strings_Vector_T is
    Ret_Val : Strings_Vector_T;
    Cursor  : Maps.Cursor := Map.First;
    To_Append : Unbounded_String;
  begin
    while Cursor /= Maps.No_Element loop
      To_Append := Maps.Key (Cursor);
      Ret_Val.Append (To_Append);
      exit when Cursor = Map.Last;
      Cursor := Maps.Next (Cursor);
    end loop;
    return Ret_Val;
  end Get_Keys;
end City_Trivia;
```

Containers Lab Solution - Main

```
with Ada.Strings.Unbounded; use Ada.Strings.Unbounded;
with Ada.Text_IO;           use Ada.Text_IO;
with City_Trivia;
procedure Main is
  Trivia : City_Trivia.Strings_Vector_T;
  Keys   : City_Trivia.Strings_Vector_T;

  function Get (Prompt : String) return String is
  begin
    Put (Prompt & "> ");
    return Get_Line;
  end Get;

begin

  Outer_Loop :
  loop
    declare
      City : constant String := Get ("City name");
    begin
      exit Outer_Loop when City'Length = 0;
      Inner_Loop :
      loop
        declare
          Info : constant String := Get ("  Trivia");
        begin
          exit Inner_Loop when Info'Length = 0;
          City_Trivia.Add_Trivia (City        => City,
                                  Information => Info);
        end;
      end loop Inner_Loop;
    end;
  end loop Outer_Loop;

  Keys := City_Trivia.Get_Keys;
  City_Trivia.Sort.Sort (Keys);
  for Key of Keys loop
    Trivia := City_Trivia.Get_Trivia (To_String (Key));
    Put_Line (To_String (Key));
    for Info of Trivia loop
      Put_Line ("   " & To_String (Info));
    end loop;
  end loop;
end Main;
```

Summary

Containers Review

- Containers class is the ultimate "code re-use"
  - Solidifies most common containers used in coding
  - Full functionality
    - When writing your own, you may not create all the functions someone else neds
  - Part of the language, so reliability is much higher
- Availability depends on language-version and runtime
  - Typically not available on certified runtimes (e.g. ravenscar)

Elaboration

Introduction

Why Elaboration Is Needed

- Ada has some powerful features that require initialization:

```
with Dep1;
package P1 is
   -- value not known at compile time
   Val : constant Integer := Dep1.Call;
end P1;
```

- May also involve dynamic allocation:

```
with P1;
package P2 is
   -- size not known at compile time
   Buffer : String (1 .. P1.Val);
end P1;
```

- Or explicit user code to initialize a package

```
package body P3 is
   ...
begin
   Put_Line ("Starting P3");
end P3;
```

- Requires initialization code at startup
- Implies ordering

Elaboration

Examples

```
with Initializer; use Initializer;
package Elab_1 is
   Spec_Object : Integer := Call (101);
   procedure Proc;
end Elab_1;

package body Elab_1 is
   Body_Object : Integer := Call (102);
   procedure Proc is null;
begin
   Body_Object := Body_Object + Call (103);
end Elab_1;

with Initializer; use Initializer;
package Elab_2 is
   Spec_Object : Integer := Call (201);
   procedure Proc;
end Elab_2;

package body Elab_2 is
   Body_Object : Integer := Call (202);
   procedure Proc is null;
begin
   Body_Object := Body_Object + Call (203);
end Elab_2;

with Elab_2;
with Elab_1;
procedure Test_Elab is
begin
   Elab_2.Proc;
   Elab_1.Proc;
end Test_Elab;

package Initializer is
   function Call (I : Integer) return Integer;
end Initializer;

with Ada.Text_IO; use Ada.Text_IO;
package body Initializer is
   function Call (I : Integer) return Integer is
   begin
      Put_Line ("Call with " & Integer'Image (I));
      return I;
   end Call;
end Initializer;
```

Elaboration

- Process where entities are created
- The Rule: "an entity has to be elaborated before use"
  - Subprograms have to be elaborated before being called
  - Variables have to be elaborated before being referenced
- Such elaboration issues typically arise due to:
  - Global variable initialization
  - Package sequence of statements

```
with Dep1;
package P1 is
   -- Dep1 body has to be elaborated before this point
   V_Spec : Integer := Dep1.Call;
end P1;

with Dep2;
package body P1 is
   V_Body : Integer;
begin
   -- Dep2 body has to be elaborated before this point
   V_Body := Dep2.Call;
end P1;
```

Elaboration Order

- The elaboration order is the order in which the packages are created
- It may or may not be deterministic

```
package P1 is
   V_Spec : Integer := Call(1);
end P1;
package body P1 is
   V_Body : Integer := Call(2);
end P1;
package P2 is
   V_Spec : Integer := Call('A');
end P1;
package body P2 is
   V_Body : Integer := Call('B');
end P1;
```

- The binder (GNAT: gnatbind) is responsible for finding an elaboration order
  - Computes the possible order
  - Reports an error when no order is possible

Circular Elaboration Dependencies

- Although not explicitly specified by the `with` clauses, elaboration dependencies may exhibit circularities
- Sometimes, they are static

```
package body P1 is
   V_Body : Integer := P2.Call;
end P1;
package body P2 is
   V_Body : Integer := P1.Call;
end P2;
```

- Sometimes they are dynamic

```
package body P1 is
   V_Body : Integer;
begin
   if Something then
      V_Body := P1.Call;
   end if;
end P1;
package body P2 is
   V_Body : Integer;
begin
   if Something then
      V_Body := P2.Call;
   end if;
end P2;
```

GNAT Static Elaboration Model

- By default, GNAT ensures elaboration safety
  - It adds elaboration control pragma to statically ensure that elaboration is possible
  - Very safe, but...
  - Not fully Ada compliant (may reject some valid programs)
  - Highly recommended however (least surprising effect)
- Performed by `gnatbind`
  - Automatically called by a builder (`gprbuild`)
  - Reads ALI files from the closure
  - Generates `b_xxx.ad[sb]` or `b__xxx.ad[sb]` files
  - Contains elaboration and finalization procedures
  - Defines the entry point procedure, **main()**.

Quiz

```
with Ada.Text_IO; use Ada.Text_IO;
package P is
   function Call (X : Integer) return Integer;
end P;
package body P is
   function Call (X : Integer) return Integer is
   begin
      Put_Line ("Call " & X'Image);
      return X;
   end Call;
end P;

with P; use P;
package P1 is
   P1_Spec : Integer := P.Call (101);
   procedure P1_Proc;
end P1;
package body P1 is
   P1_Body : Integer := P.Call (102);
   procedure P1_Proc is null;
end P1;

with P; use P;
package P2 is
   P2_Spec : Integer := P.Call (201);
   procedure P2_Proc;
end P2;
package body P2 is
   P2_Body : Integer := P.Call (202);
   procedure P2_Proc is null;
end P2;

with P2; with P1;
procedure Main is
begin
   null;
end Main;
```

What is the output of running this program

A. 101, 102, 201, 202
B. 201, 202, 101, 102
C. 101, 201, 102, 202
D. Cannot be determined

Quiz

```
with Ada.Text_IO; use Ada.Text_IO;
package P is
   function Call (X : Integer) return Integer;
end P;
package body P is
   function Call (X : Integer) return Integer is
   begin
      Put_Line ("Call " & X'Image);
      return X;
   end Call;
end P;

with P; use P;
package P1 is
   P1_Spec : Integer := P.Call (101);
   procedure P1_Proc;
end P1;
package body P1 is
   P1_Body : Integer := P.Call (102);
   procedure P1_Proc is null;
end P1;

with P; use P;
package P2 is
   P2_Spec : Integer := P.Call (201);
   procedure P2_Proc;
end P2;
package body P2 is
   P2_Body : Integer := P.Call (202);
   procedure P2_Proc is null;
end P2;

with P2; with P1;
procedure Main is
begin
   null;
end Main;
```

What is the output of running this program

A. 101, 102, 201, 202
B. 201, 202, 101, 102
C. 101, 201, 102, 202
D. ***Cannot be determined***

As there are no dependencies between `P1` and `P2`, the compiler/linker can enforce any elaboration order. Even the order of `with`'s" in `Main` may not affect elaboration order

Elaboration Control

Examples

```
package Pure_P is
   pragma Pure;
   Some_Constant : constant Integer := Integer'Size;
   function Call (I : Integer) return Integer is (I);
end Pure_P;

with Pure_P;
package Preelaborate_P is
   pragma Preelaborate;
   Global_Object : Integer := Pure_P.Some_Constant;
end Preelaborate_P;

package Elaborate_Body_P is
   pragma Elaborate_Body;
   function Call (I : Integer) return Integer;
end Elaborate_Body_P;

with Ada.Text_IO; use Ada.Text_IO;
package body Elaborate_Body_P is
   function Call (I : Integer) return Integer is
   begin
      Put_Line ("Call with " & Integer'Image (I));
      return I;
   end Call;
begin
   Put_Line ("Elaborate_Body_P package execution");
end Elaborate_Body_P;

with Elaborate_Body_P; use Elaborate_Body_P;
pragma Elaborate (Elaborate_Body_P);
package Elab_1 is
   Spec_Object : Integer := Call (101);
   procedure Proc;
end Elab_1;

with Elab_2;
package body Elab_1 is
   Body_Object : Integer := Call (102);
   procedure Proc is null;
begin
   Body_Object := Body_Object + Call (103);
   Elab_2.Proc;
end Elab_1;

with Elaborate_Body_P; use Elaborate_Body_P;
package Elab_2 is
   Spec_Object : Integer := Call (201);
   procedure Proc;
end Elab_2;

package body Elab_2 is
   Body_Object : Integer := Call (202);
   procedure Proc is null;
begin
   Body_Object := Body_Object + Call (203);
end Elab_2;

with Elab_2;
with Elab_1;
pragma Elaborate_All (Elab_2);
procedure Test_Elab_Control is
begin
   Elab_1.Proc;
   Elab_2.Proc;
end Test_Elab_Control;
```

Pragma Preelaborate

- Adds restrictions on a unit to ease elaboration
- Elaboration without explicit execution of code
  - No user initialization code
  - No calls to subprograms
  - Static values
  - Dependencies only on **Preelaborate** packages

```
package P1 is
   pragma Preelaborate;
   Var : Integer := 7;
end P1;
```

- But compiler may generate elaboration code

```
package P1 is
  pragma Preelaborate;
  type Ptr is access String;
  v : Ptr := new String'("hello");
end P1;
```

Pragma Pure

- Adds more restrictions on a unit to ease elaboration
- **Preelaborate** restrictions plus ...
  - No variable declaration
  - No allocators
  - No access type declaration
  - Dependencies only on **Pure** packages

```
package Ada.Numerics is
   pragma Pure;
   Argument_Error : exception;
   Pi : constant := 3.14...;
end Ada.Numerics;
```

- But compiler may generate elaboration code

```
package P2 is
   pragma Pure;
   Var : constant Array (1 .. 10 * 1024) of Integer :=
         (others => 118);
end P2;
```

Pragma Elaborate_Body

- Forces the elaboration of a body just after a specification
- Forces a body to be present even if none is required
- Problem: it may introduce extra circularities

```
package P1 is
   pragma Elaborate_Body;
   function Call return Integer;
end P1;
with P2;
package body P1 is
..
end P1;
package P2 is
   pragma Elaborate_Body;
   function Call return Integer;
end P2;
with P1;
package body P2 is
...
end P2;
```

- Useful in the case where a variable declared in the specification is initialized in the body

Pragma Elaborate

- `Pragma Elaborate` forces the elaboration of a dependency body
- It does not force the elaboration of transitive dependencies

```
package P1 is
   function Call return Integer;
end P1;
package P2 is
   function Call return Integer;
end P1;
with P1;
package body P2 is
   function Call return Integer is
   begin
      return P1.Call;
   end Call;
end P2;
with P2;
pragma Elaborate (P2);
-- P2 must be elaborated before we get here
-- but nobody forces P1 to be elaborated!
package body P3 is
   V : Integer;
begin
   V := P2.Call;
end P3;
```

Pragma Elaborate_All

- `Pragma Elaborate_All` forces the elaboration of a dependency body and all transitive dependencies
- May introduce unwanted cycles
- Safer than **Elaborate**

```
package P1 is
   function Call return Integer;
end P1;
package P2 is
   function Call return Integer;
end P1;
with P1;
package body P2 is
   function Call return Integer is
   begin
      return P1.Call;
   end Call;
end P2;
with P2;
pragma Elaborate_All (P2);
-- P2 must be elaborated before we get here.
-- Elaborate_All enforces P1 being elaborated as well
package body P3 is
   V : Integer;
begin
   V := P2.Call;
end P3;
```

Lab

Elaboration Lab

- Requirements
  - Create a **pure** package containing some constants
    - Lower limit of some integer range
    - Upper limit of some integer range
    - Flag indicating an invalid state
  - Create a package whose interface consists solely of one global object
    - Array of integers initialized to the invalid state
  - During elaboration, fill in the array object by querying the user
    - All entries must be in the range of *Lower Limit* to *Upper Limit*
  - Create a **main** program to print out the array
    - Only print values set by the user
- Hints
  - The only indication of actual number of entries is the array itself
  - Need to tell the compiler that the global object is initialized in the package body

Elaboration Lab Solution - Constants

```
package Constants is
   pragma Pure;

   Minimum_Value : constant := -1_000;
   Maximum_Value : constant := 15_000;
   Invalid_Value : constant := Integer'Last;

end Constants;
```

Elaboration Lab Solution - Data Store

```
package Datastore is
  pragma Elaborate_Body;

  Object : array (1 .. 100) of Integer;

end Datastore;

with Constants;
with Ada.Text_IO; use Ada.Text_IO;
package body Datastore is

  subtype Valid_Range is
   Integer range Constants.Minimum_Value .. Constants.Maximum_Value;
  Attempt : Integer;
  Count   : Integer := Object'First;

begin

  loop
    Put ("Value: ");
    Attempt := Integer'Value (Ada.Text_IO.Get_Line);
    exit when Attempt not in Valid_Range;
    Object (Count) := Attempt;
    Count          := Count + 1;
  end loop;

  for I in Count .. Object'Last loop
    Object (I) := Constants.Invalid_Value;
  end loop;

end Datastore;
```

Elaboration Lab Solution - Main

```
with Ada.Text_IO; use Ada.Text_IO;
with Constants;
with Datastore;
procedure Main is

begin

   for I in Datastore.Object'First .. Datastore.Object'Last
   loop
      exit when Datastore.Object (I) = Constants.Invalid_Value;
      Put_Line (Integer'Image (I) & " =>" &
                Integer'Image (Datastore.Object (I)));
   end loop;

end Main;
```

Summary

Summary

- Elaboration is a difficult problem to deal with
- The binder tries to resolve it in a "safe way"
- If it can't, it's possible to manually place elaboration pragmas
- Better to avoid elaboration constraints as much as possible
- Use dynamic elaboration (gnat binder switch -E) as last resort
- See *Elaboration Order Handling in GNAT* annex in GNAT Pro User's Guide.

Controlled Types

Introduction

Constructor / Destructor

- Possible to specify behavior of object initialization, finalization, and assignment
  - Based on type definition
  - Type must derive from **Controlled** or **Limited_Controlled** in package **Ada.Finalization**
- This derived type is called a *controlled type*
  - User may override any or all subprograms in **Ada.Finalization**
  - Default implementation is a null body

Ada.Finalization

Package Spec

```
package Ada.Finalization is

   type Controlled is abstract tagged private;
   procedure Initialize(Object : in out Controlled)
      is null;
   procedure Adjust    (Object : in out Controlled)
      is null;
   procedure Finalize  (Object : in out Controlled)
      is null;

   type Limited_Controlled is abstract tagged limited private;
   procedure Initialize(Object : in out Limited_Controlled)
      is null;
   procedure Finalize  (Object : in out Limited_Controlled)
      is null;

private
   -- implementation defined
end Ada.Finalization;
```

Uses

- Prevent "resource leak"
  - Logic centralized in service rather than distributed across clients
- Examples: heap reclamation, "mutex" unlocking
- User-defined assignment

Initialization

- Subprogram **Initialize** invoked after object created
  - Either by object declaration or allocator
  - Only if no explicit initialization expression
- Often default initialization expressions on record components are sufficient
  - No need for an explicit call to **Initialize**
- Similar to C++ constructor

Finalization

- Subprogram **Finalize** invoked just before object is destroyed
  - Leaving the scope of a declared object
  - Unchecked deallocation of an allocated object
- Similar to C++ destructor

Assignment

- Subprogram **Adjust** invoked as part of an assignment operation
- Assignment statement **Target := Source;** is basically:
  - **Finalize (Target)**
  - Copy Source to Target
  - **Adjust (Target)**
  - *Actual rules are more complicated, e.g. to allow cases where Target and Source are the same object*
- Typical situations where objects are access values
  - **Finalize** does unchecked deallocation or decrements a reference count
  - The copy step copies the access value
  - **Adjust** either clones a "deep copy" of the referenced object or increments a reference count

Example

Unbounded String via Access Type

- Type contains a pointer to a string type
- We want the provider to allocate and free memory "safely"
  - No sharing
  - **Adjust** allocates referenced String
  - **Finalize** frees the referenced String
  - Assignment deallocates target string and assigns copy of source string to target string

Unbounded String Usage

```
with Unbounded_String_Pkg; use Unbounded_String_Pkg;
procedure Test is
   U1 : Ustring_T;
begin
   U1 := To_Ustring_T ("Hello");
   declare
      U2 : Ustring_T;
   begin
      U2 := To_Ustring_T ("Goodbye");
      U1 := U2; -- Reclaims U1 memory
   end; -- Reclaims U2 memory
end Test; -- Reclaims U1 memory
```

Unbounded String Definition

```
with Ada.Finalization; use Ada.Finalization;
package Unbounded_String_Pkg is
   -- Implement unbounded strings
   type Ustring_T is private;
   function "=" (L, R : Ustring_T) return Boolean;
   function To_Ustring_T (Item : String) return Ustring_T;
   function To_String (Item : Ustring_T) return String;
   function Length (Item : Ustring_T) return Natural;
   function "&" (L, R : Ustring_T) return Ustring_T;
private
   type String_Ref is access String;
   type Ustring_T is new Controlled with record
      Ref : String_Ref := new String (1 .. 0);
   end record;
   procedure Finalize (Object : in out Ustring_T);
   procedure Adjust (Object : in out Ustring_T);
end Unbounded_String_Pkg;
```

Unbounded String Implementation

```
with Ada.Unchecked_Deallocation;
package body Unbounded_String_Pkg is
   procedure Free_String is new Ada.Unchecked_Deallocation
     (String, String_Ref);

   function "=" (L, R : Ustring_T) return Boolean is
      ( L.Ref.all = R.Ref.all );

   function To_Ustring_T (Item : String) return Ustring_T is
      ( Controlled with Ref => new String'(Item) );

   function To_String (Item : Ustring_T) return String is
      ( Item.Ref.all );

   function Length (Item : Ustring_T) return Natural is
      ( Item.Ref.all'Length );

   function "&" (L, R : Ustring_T) return Ustring_T is
      (Controlled with Ref => new String'(L.Ref.all & R.Ref.all);

   procedure Finalize (Object : in out Ustring_T) is
   begin
      Free_String (Object.Ref);
   end Finalize;

   procedure Adjust (Object : in out Ustring_T) is
   begin
      Object.Ref := new String'(Object.Ref.all);
   end Adjust;
end Unbounded_String_Pkg;
```

Lab

Controlled Types Lab

- Requirements
  - Create a simplistic secure key tracker system
    - Keys should be unique
    - Keys cannot be copied
    - When a key is no longer in use, it is returned back to the system
  - Interface should contain the following methods
    - Generate a new key
    - Return a generated key
    - Indicate how many keys are in service
    - Return a string describing the key
  - Create a main program to generate / destroy / print keys
- Hints
  - Need to return a key when out-of-scope OR on user request
  - Global data to track used keys

Controlled Types Lab Solution - Keys (Spec)

```
with Ada.Finalization;
package Keys_Pkg is

   type Key_T is limited private;
   function Generate return Key_T;
   procedure Destroy (Key : Key_T);
   function In_Use return Natural;
   function Image (Key : Key_T) return String;

private
   type Key_T is new Ada.Finalization.Limited_Controlled with record
      Value : Character;
   end record;
   procedure Initialize (Key : in out Key_T);
   procedure Finalize (Key : in out Key_T);

end Keys_Pkg;
```

Controlled Types Lab Solution - Keys (Body)

```
package body Keys_Pkg is
   Global_In_Use : array (Character range 'a' .. 'z') of Boolean :=
     (others => False);

   pragma Warnings ( Off );
   function Next_Available return Character is
   begin
      for C in Global_In_Use'Range loop
         if not Global_In_Use (C) then
            return C;
         end if;
      end loop;
      -- we ran out of keys! exception if we get here
   end Next_Available;
   pragma Warnings ( On );

   function In_Use return Natural is
      Ret_Val : Natural := 0;
   begin
      for Flag of Global_In_Use loop
         Ret_Val := Ret_Val + (if Flag then 1 else 0);
      end loop;
      return Ret_Val;
   end In_Use;

   function Generate return Key_T is
   begin
      return X : Key_T;
   end Generate;

   procedure Destroy (Key : Key_T) is
   begin
      Global_In_Use (Key.Value) := False;
   end Destroy;

   function Image (Key : Key_T) return String is
      ( "KEY: " & Key.Value );

   procedure Initialize (Key : in out Key_T) is
   begin
      Key.Value                 := Next_Available;
      Global_In_Use (Key.Value) := True;
   end Initialize;

   procedure Finalize (Key : in out Key_T) is
   begin
      Global_In_Use (Key.Value) := False;
   end Finalize;
end Keys_Pkg;
```

Controlled Types Lab Solution - Main

```
with Keys_Pkg;
with Ada.Text_IO; use Ada.Text_IO;
procedure Main is

   procedure Generate (Count : Natural) is
      Keys : array (1 .. Count) of Keys_Pkg.Key_T;
   begin
      Put_Line ("In use: " & Integer'Image (Keys_Pkg.In_Use));
      for Key of Keys
      loop
         Put_Line ("   " & Keys_Pkg.Image (Key));
      end loop;
   end Generate;

begin
   Put_Line ("In use: " & Integer'Image (Keys_Pkg.In_Use));

   Generate (4);
   Put_Line ("In use: " & Integer'Image (Keys_Pkg.In_Use));

end Main;
```

Summary

Summary

- Controlled types allow access to object construction, assignment, destruction
- **Ada.Finalization** can be expensive to use
  - Other mechanisms may be more efficient
    - But require more rigor in usage