An Introduction to Ada's Predefined Packages

The Annexes of the Ada Reference Manual (ARM) provides descriptions of the packages that every conforming Ada compiler vendor must provide (Chapters 1-13 and Annex A) and those that they usually provide. Here is a list of the names of those packages along with the location of their specifications in the ARM.

Standard -- A.1
Ada -- A.2
Asynchronous_Task_Control -- D.11
Calendar -- 9.6
Characters -- A.3.1
Handling -- A.3.2
Latin_1 -- A.3.3
Command_Line -- A.15
Decimal -- F.2
Direct_IO -- A.8.4
Dynamic_Priorities -- D.5
Exceptions -- 11.4.1
Finalization -- 7.6
Interrupts -- C.3.2
Names -- C.3.2
IO_Exceptions -- A.13
Numerics -- A.5
Complex_Elementary_Functions -- G.1.2
Complex_Types -- G.1.1
Discrete_Random -- A.5.2
Elementary_Functions -- A.5.1
Float_Random -- A.5.2
Generic_Complex_Elementary_Functions -- G.1.2
Generic_Complex_Types -- G.1.1
Generic_Elementary_Functions -- A.5.1
Real_Time -- D.8
Sequential_IO -- A.8.1
Storage_IO -- A.9
Streams -- 13.13.1
Stream_IO -- A.12.1
Strings -- A.4.1
Bounded -- A.4.4
Fixed -- A.4.3
Maps -- A.4.2
Constants -- A.4.6

Strings (continued)
Unbounded -- A.4.5
Wide_Bounded -- A.4.7
Wide_Fixed -- A.4.7
Wide_Maps -- A.4.7
Wide_Constants -- A.4.7
Wide_Unbounded -- A.4.7
Synchronous_Task_Control -- D.10
Tags -- 3.9
Task_Attributes -- C.7.2
Task_Identification -- C.7.1
Text_IO -- A.10.1
Complex_IO -- G.1.3
Editing -- F.3.3
Text_Streams -- A.12.2
Unchecked_Conversion -- 13.9
Unchecked_Deallocation -- 13.11.2
Wide_Text_IO -- A.11
Complex_IO -- G.1.3
Editing -- F.3.4
Text_Streams -- A.12.3

Interfaces -- B.2
C -- B.3
Pointers -- B.3.2
Strings -- B.3.1
COBOL -- B.4
Fortran -- B.5

System -- 13.7
Address_To_Access_Conversions -- 13.7.2
Machine_Code -- 13.8
RPC -- E.5
Storage_Elements -- 13.7.1
Storage_Pools -- 13.11

The purpose of this guide is to help you learn how to read the specifications for these packages. This help is in the form of more detailed descriptions and examples of some of the more commonly used resources in the character, string, and numerics packages. Once you can read these specifications, you will have an easier time reading the others.

While many of the descriptions and examples in this guide stand alone, it is recommended that you have a copy of the ARM nearby. AdaGIDE includes a copy of the ARM in its help section.
Character Handling
There are two different character types in Ada 95:

- **Character**: 256 different characters specified by ISO 8859-1 (also called Latin-1).
- **Wide_Character**: 65,536 different characters specified by ISO 10646.

Character literals for both of these types follow the same syntax. 'A' could be either a Character literal or a Wide_Character literal. Usually we (and the Ada compiler) can tell the type of a literal from how it is used. For those few times that the compiler cannot tell the difference we must qualify the literal as in `Character'('A')`.

**Package Ada.Characters.Handling (ARM A.3.2)**
This package contains useful functions for dealing with characters and strings. Here are what some of them do.

**Classification Functions**
A set of Boolean functions used to determine classes of characters. You can ask whether or not a particular character is:

- A control character (one that controls the output device)
- A graphics character (one that can be displayed on an output device)
- A letter (one of the 26 letters of the alphabet)
- A lowercase letter
- An uppercase letter
- A digit (one of the ten decimal digits)
- A hexadecimal digit (one of the sixteen hexadecimal digits)
- An alphanumeric character (one that is either a letter or a digit)
- A special character (one that is a graphic character and not alphanumeric)

**Conversion Functions**
A set functions used to convert between lowercase and uppercase. There are functions for both individual characters and strings.

A set of functions to convert between Character and Wide_Character. There are functions for both individual characters and strings.

A set of functions to convert between Character and ASCII. There are functions for both individual characters and strings.

**Package Ada.Characters.Latin_1 (ARM A.3.3)**
This package contains named constants for all the non graphics characters and many graphics characters not commonly found on a keyboard. For example, the following displays ©

```
Ada.Text_IO.Put (Item => Ada.Characters.Latin_1.Copyright_Sign);
```
String Handling

The three common string types are

- **Fixed Length Strings**: A string whose length (number of characters) is constant
- **Bounded Length Strings**: A variable length string with an upper bound on its length
- **Unbounded Length Strings**: A variable length string with no upper bound on its length

Ada provides support for each of these string types for both character strings and wide character strings. This document describes the more commonly used character string packages. The wide character string packages are identical.

**Package Ada.Strings** *(ARM A.4.1)*

This package contains constants, enumeration types, and exception names used by the other string packages. Here is part of this package.

```ada
package Ada.Strings is
  Space : constant Character := ' ';                    -- These exceptions are
  Length_Error      : exception;                       -- raised by various
  Pattern_Error     : exception;                      -- operations in the
  Index_Error       : exception;                      -- predefined string packages
  Translation_Error : exception;                      -- set difference (in first set but not in second set)

  type Alignment  is (Left, Right, Center);           -- What characters to drop when the string won't fit?
  type Truncation is (Left, Right, Error);            -- Select elements that are inside or outside of a set
  type Membership is (Inside, Outside);               -- For searching a string for a pattern
  type Direction  is (Forward, Backward);             -- End of string from which to trim pad characters?
  type Trim_End   is (Left, Right, Both);             -- subset

end Ada.Strings;
```

**Package Ada.Strings.Maps** *(ARM A.4.2)*

This package contains constants and functions used by the other string packages. This package defines three abstract data types.

1. A **Sequence** of characters. This is a synonym for the standard type String.

2. A **Set** whose elements are characters. Operations include
   a) Standard set operations including
      - **or** union of two character sets
      - **and** intersection of two character sets
      - **-** set difference (in first set but not in second set)
      - **xor** in one of the sets but not in both sets
      - **not** a set all characters not in the set
      - **ls_in** set membership
      - **<=** subset
b) Constructors to build a set.

The simplest constructor creates a set of one character from a character. Here is an example of using this constructor to create a set containing only the character P.

```
My_Set := Ada.Strings.Maps.To_Set (Singleton => 'P');
```

A more useful constructor creates a set from a sequence of characters. Here is an example of using this constructor to create a set of vowels (with 10 characters in the set).

```
My_Set := Ada.Strings.Maps.To_Set (Sequence => "AEIOUaeiou");
```

Other character set constructors make use of one or more ranges of characters. Each range is defined by a record containing the upper and lower characters in the range. Here is an example of one of these function calls. It creates a set of 26 uppercase letters. The actual parameter in this example is a record aggregate.

```
My_Set := Ada.Strings.Maps.To_Set (Span => ('A', 'Z'));
```


Mappings are used to translate one character into another. An example use of a mapping is the conversion of an uppercase letter to a lowercase letter.

The following translates the character 'P' into another character as determined by the mapping in the variable My_Map.

```
Translated_Char := Ada.Strings.Maps.Value (Map     => My_Map,
                                          Element => 'P');
```

The function To_Mapping constructs a map from two character sequences. For example, the following function call creates a mapping that can be used to convert vowels into digits. The characters 'E' and 'e' are both mapped to the character '2'.

```
My_Map := Ada.Strings.Maps.To_Mapping (From => "AEIOUaeiou",
                                        To   => "1234512345");
```

Characters not in the From parameter are not mapped. Thus after setting up My_Map to translate the five vowels, the value of 'P' returned by the Value function will be 'P'.

The package defines the named constant `identity` which is a mapping that maps each character to itself.


This package contains 14 predefined character sets and mappings such as a set of lowercase letters and a mapping from uppercase to lowercase.
Package Ada.Strings.Fixed  (ARM A.4.3)

This package provides 31 functions and procedures for working with the standard type String. These operations are supplied as additions to the standard operations available for Strings (e.g. assignment, relational operations, indexing, slicing, etc.).

This document does not describe all 31 operations defined in the package. It provides more substantial documentation than provided in the Ada Reference Manual (ARM) for a few of the more important operations. Understanding these operations will allow you to more easily read the ARM documentation for the remaining operations.

Standard Strings contain a fixed number of characters. Programmers often use the model that a String variable contains significant characters and possible padding (usually space characters). Let's look at an example.

```ada
subtype Name_String is String (1..10);
Name : Name_String;  -- A string of 10 characters
Name := "Mildred   ";  -- Assign 7 significant characters and 3 spaces
```

Padding can be added on the right (as above), on the left, or to both sides of the string to center the significant characters.

Copying Strings

Problems arise when copying fixed length strings that contain different numbers of characters. Operations in package Ada.Strings.Fixed follow these two guidelines:

- When a shorter string is copied to a larger string, padding is inserted.
- When a larger string is copied to a smaller string, padding is stripped. Significant characters may also be stripped if removing all the padding does not reduce the number of characters in the longer string to what the shorter string can hold.

The procedure in the package for copying standard Strings is Move. Here is its specification:

```ada
procedure Move (Source : in  String;
Target : out String;
Drop    : in  Truncation := Error;
-- These types are defined in
Justify : in  Alignment := Left;
-- the package Ada.Strings
Pad     : in  Character  := Space);
```

Procedure Move puts a copy of Source into Target. We use the parameters Drop, Justify, and Pad to describe what we want the Move procedure to do if Source and Target are different size strings.

- **Pad**
  - Tells what character is stripped or added when copying strings of different sizes
  - The default value is the space character.
An Introduction to Ada's Predefined Packages

**Justify**
Tells how to justify the significant characters in **Target**. The default is to left justify the significant characters by stripping from or adding padding on the right.

**Drop**
Tells from which end to strip significant characters if necessary. The default is to raise the exception **Length_Error** if the significant characters in **Source** won't all fit in **Target**.

Here are some examples of calling the Move procedure. To save space on this page we did not include any qualification information. You will need to qualify many identifiers with their package names in your programs.

```ada
subtype Name_String is String (1..10);
subtype Short_String is String (1..4);

Name : Name_String;  -- Holds 10 characters
Short : Short_String; -- Holds 4 characters

Move (Source => "Horace", -- Name becomes "Horace    " which includes
    Target  => Name), -- 4 padding (space) characters after the e
    Justify => Center);

Move (Source => "Horace", -- Name becomes "Horace    " which includes
    Target  => Name), -- 4 padding (space) characters after the e
    Justify => Center);

Move (Source => "Horace", -- Name becomes "Horace    " which includes
    Target  => Name), -- 4 padding (space) characters after the e
    Justify => Center);

Move (Source => "Horace", -- Name becomes "$\$Horace\$\$"
    Target  => Name),
    Justify => Center,
    Pad     => '$');

Move (Source => "Horace", -- Exception **Length_Error** raised
    Target  => Short);

Move (Source => "Horace", -- Short becomes "Hora"
    Target  => Short),
    Drop    => Right);

Move (Source => "Horace", -- Short becomes "race"
    Target  => Short),
    Drop    => Left);
```

It is important to understand the exact behavior of the Move procedure as many of the operations in package Ada.Strings.Fixed are described in terms of calls to procedure Move.
Searching Strings

Package Ada.Strings.Fixed contains a number of different functions for searching a standard String. We discuss three of these. There are more in the package.

The following function searches for a substring within a string. It returns the location (the index) of the first character of a substring (Pattern) within a string (Source). If Pattern does not exist within Source, a value of zero is returned.

```ada
function Index (Source   : in String;
                Pattern  : in String;
                Going    : in Direction := Forward;  -- Type defined in Ada.Strings
          return Natural;
```

Here are some example calls that use the default parameter values:

- `Location := Ada.Strings.Fixed.Index (Source  => "Mildred", Pattern => "Red");`  -- Returns 0
- `Location := Ada.Strings.Fixed.Index (Source  => "Mildred", Pattern => "d");`  -- Returns 4

The parameters Going and Mapping give us additional control over the search.

Going  Tells what direction to perform the search. The default is a forward search (left to right).

Mapping Tells how to translate characters in the Pattern before comparing the Pattern to the Source. The default is to do no translation. Package Ada.Strings.Maps contains the declarations and operations for character mapping.

Here are some more examples with values supplied for Going and Mapping.

- `-- Create a mapping (Could use Ada.Strings.Maps.Constants.Lower_Case_Map)
  Upper_To_Lower := Ada.Strings.Maps.To_Mapping
    (From => "ABCDEFGHIJKLMNOPQRSTUVWXYZ",
     To   => "abcdefghijklmnopqrstuvwxyz");
  Location := Ada.Strings.Fixed.Index (Source  => "Mildred", Pattern => "Red", Mapping => Upper_To_Lower);`  -- Returns 5

The exception Pattern_Error is raised if Pattern is a null string (contains zero characters).
Our second search function searches for one of several characters within a string. It returns the location (the index) of the first character it finds in \textit{Source} that is in \textit{Set}. If it can't find one of the set elements in the string, it returns a value of zero.

\begin{verbatim}
function Index (Source : in String;
    Set    : in Maps.Character_Set;
    Test   : in Membership := Inside;  -- These types are defined in
    Going  : in Direction := Forward)  -- the package Ada.Strings
return Natural;
\end{verbatim}

Here are some example calls:

\begin{verbatim}
-- Create a set of vowels
Vowel_Set := Ada.Strings.Maps.To_Set (Sequence => "AEIOUaeiou");

-- Find first vowel in the string
Location := Ada.Strings.Fixed.Index (Source  => "Mildred",      -- Returns 2
                                     Set     => Vowel_Set);

-- Find first vowel in the string
Location := Ada.Strings.Fixed.Index (Source  => "Mnnppl",       -- Returns 0
                                     Set     => Vowel_Set);
\end{verbatim}

The parameters \textit{Going} and \textit{Test} give us additional control over the search.

\textbf{Going}  
Tells what direction to perform the search. The default is a forward search (left to right).

\textbf{Test}  
Tells whether to search for characters that are inside the set or outside the set.

Here are some more examples with values supplied for \textit{Going} and \textit{Test}.

\begin{verbatim}
-- Find last vowel in the string
Location := Ada.Strings.Fixed.Index (Source  => "Mildred",      -- Returns 6
                                     Set     => Vowel_Set,
                                     Going   => Ada.Strings.Backward);

-- Find first non-vowel in the string
Location := Ada.Strings.Fixed.Index (Source  => "Mildred",      -- Returns 1
                                     Set     => Vowel_Set,
                                     Test    => Ada.Strings.Outside);
\end{verbatim}
Our last search procedure is similar to the previous function. Instead of searching for one of several characters within a string, it looks for the first contiguous sequence of these characters. It returns the location (the index) of the First and Last character in the sequence it finds in Source. If it can't find such a group of set elements in the string, it returns a values of Source'First for First and zero for Last.

```ada
procedure Find_Token (Source : in String;
                     Set    : in Maps.Character_Set;
                     Test   : in Membership; -- Type defined in Ada.Strings
                     First  : out Positive;
                     Last   : out Natural);
```

Here are some example calls:

```ada
-- Create a set of vowels
Vowel_Set := Ada.Strings.Maps.To_Set (Sequence => "AEIOUaeiou");

-- The following call returns 2 and 4 for First and Last
Ada.Strings.Fixed.Find_Token (Source => "Beautiful",
                              Set   => Vowel_Set,
                              Test  => Ada.Strings.Inside,
                              First => Start_Of_Vowel_String,
                              Last  => End_Of_Vowel_String);

-- The following call returns 1 and 0 for First and Last
Ada.Strings.Fixed.Find_Token (Source => "Mnppls",
                              Set   => Vowel_Set,
                              Test  => Ada.Strings.Inside,
                              First => Start_Of_Vowel_String,
                              Last  => End_Of_Vowel_String);

-- Create a set with a single element - the space character
Blank_Set := Ada.Strings.Maps.To_Set (Singleton => ' ')

-- The following call finds the location of the first word of Source.
-- It returns 3 and 7 for First and Last
Ada.Strings.Fixed.Find_Token (Source => " Hello Mildred ",
                              Set   => Blank_Set,
                              Test  => Ada.Strings.Outside,
                              First => Start_Of_Word,
                              Last  => End_Of_Word);

-- The following call finds the location of the first word of Source.
-- It returns 1 and 7 for First and Last
Ada.Strings.Fixed.Find_Token (Source => "Mildred",
                              Set   => Blank_Set,
                              Test  => Ada.Strings.Outside,
                              First => Start_Of_Word,
                              Last  => End_Of_Word);
```
Package Ada.Strings.Bounded  (ARM A.4.4)

This package provides a number of types, constants, and 66 functions and procedures for working with bounded length character strings. There is an equivalent package for bounded length wide character strings. A bounded length string is a variable length string. Unlike the fixed length strings of the previous section, the number of characters in a variable bounded length string can change over time from a minimum of zero (a null string) to some maximum value.

This is a generic package which we instantiate with a value to specify the upper bound on the number of characters a string variable can hold. Here is an example instantiation that creates a bounded length string that holds a maximum of ten characters. Following is a use type statement to make all of the operators (such as "<" and "&") in the new package available without qualification.

package Names is new Ada.Strings.Bounded.Generic_Bounded_Length (Max => 10);  
use type Names.Bounded_String;

Length

The length function returns the number of characters in a bounded length string.

function Length (Source : in Bounded_String) return Length_Range;

Here is an example of its use.

First_Name : Names.Bounded_String;  
Count      : Natural;

-- Count contains the number of characters in First_Name  
Count := Names.Length (First_Name);

Conversion

The bounded length strings package contains two functions for converting between fixed length and bounded length strings.

function To_Bounded_String (Source : in String;  
                          Drop   : in Truncation := Error) return Bounded_String;

function To_String (Source : in Bounded_String) return String;

The Drop parameter in the first conversion function allows us to specify what is to be done if the number of characters in Source exceeds the upper bound of the bounded length string. We can choose to raise the exception Length_Error (the default) or drop enough characters from the left or right side of the source to fit within the bound. Type Truncation is defined in package Ada.Strings.

Here are some examples of conversion function use.
An Introduction to Ada's Predefined Packages

subtype Name_String is String (1..10); -- A fixed length string type

Fixed_Name : Name_String; -- A fixed length string
Length     : Natural;
Name       : Names.Bounded_String; -- A bounded length string

-- Fixed length to bounded length.
Name := Names.To_Bounded_String (Source => "Mildred");

-- This conversion raises Length_Error
Name := Names.To_Bounded_String (Source => "Hello Mildred");

-- This conversion puts "Hello Mild" into Name
Name := Names.To_Bounded_String (Source => "Hello Mildred",
  Drop   => Ada.Strings.Right);

-- Set up for next examples
Name := Names.To_Bounded_String (Source => "Mildred");

-- Bounded length to fixed length. Notice use of slicing.
Fixed_Name (1..7) := Names.To_String (Source => Name);

-- Usually we need to calculate the range of the slice.
Fixed_Name (1..Names.Length(Name)) := Names.To_String (Source => Name);

-- For complete control, use the fixed length string move operation
-- instead of the assignment operator.
Ada.Strings.Fixed.Move (Source  => Names.To_String (Name),
  Target  => Fixed_Name,
  Justify => Ada.Strings.Left,
  Drop    => Ada.Strings.Error,
  Pad     => ' ');

Relational Operators

Package Ada.Strings.Bounded contains relational operators for comparing strings

=  <  <=  >  >=

These operators are defined for comparing bounded length strings to bounded length strings and bounded length strings to fixed length strings.

Catenation

Package Ada.Strings.Bounded contains operators, functions, and procedures for catenating bounded length strings. There are versions to catenate two bounded strings into a bounded length string and versions to catenate a fixed length and a bounded length string into a bounded length string. As with fixed length strings, catenation operators are also available with character parameters. The functions and procedures provide a Drop parameter used to specify what to do when the combined string exceeds the bound of the bounded length string.
Here are the specifications of the catenation operators for combining two bounded length strings. The specifications for the mixed string type operations are similar.

```ada
function Append (Left  : in Bounded_String;  
                 Right : in Bounded_String;  
                 Drop  : in Truncation := Error)   return Bounded_String;

procedure Append (Source   : in out Bounded_String;    
                  New_Item : in     Bounded_String;    
                  Drop     : in     Truncation := Error);

function "&" (Left  : in Bounded_String;  
              Right : in Bounded_String)   return Bounded_String;
```

**Selection**

We use array operations (component selection and slicing) to get and change individual characters and substrings in fixed length strings. For example

```ada
Ada.Text_IO.Put (Fixed_Name(4));     -- Get and display the fourth character.
Fixed_Name(3) = 'y';                 -- Replace the third character

Ada.Text_IO.Put (Fixed_Name(1..8));  -- Get and display the first 8 characters
Fixed_Name(1..5) = "Hello";          -- Replace the first 5 characters
```

Package Ada.Strings.Bounded provides the same functionality for bounded length strings with the following subprograms:

```ada
function Element (Source : in Bounded_String;  
                  Index  : in Positive)  return Character;  -- Get a character

procedure Replace_Element (Source : in out Bounded_String;    
                           Index  : in     Positive;    
                           By     : in     Character);  -- Change a character

function Slice (Source : in Bounded_String;  
                Low    : in Positive;  
                High   : in Natural)  return String;  -- Get a slice

procedure Replace_Slice (Source : in out Bounded_String;    
                        Low    : in     Positive;    
                        High   : in     Natural;    
                        By     : in     String;    
                        Drop   : in     Truncation := Error);  -- Replace a slice
```

Package Ada.Strings.Bounded also contains the functions Head and Tail for getting slices from the beginning and ends of the string. In these functions you give the number of characters you want to slice out rather than the indices of the first and last characters in the slice.
Inserting, Deleting, and Trimming

Package Ada.Strings.Bounded provides operations to insert a string into a string and to delete a slice of a string. Other functions make trimming leading and trailing blanks (or any other characters) easy. Here are some of their specifications:

```ada
procedure Insert (Source   : in out Bounded_String; -- Insert a string into Source
                   Before   : in     Positive;
                   New_Item : in     String;
                   Drop     : in     Truncation := Error);

procedure Delete (Source  : in out Bounded_String; -- Deleted a slice from Source
                  From    : in     Positive;
                  Through : in     Natural);

procedure Trim (Source : in out Bounded_String; -- Trim off blanks from
                Side   : in     Trim_End); -- the left, right or both

procedure Trim (Source : in out Bounded_String; -- Trim off any character in the
                Left   : in     Maps.Character_Set; -- sets from the left and
                Right  : in     Maps.Character_Set); -- right sides of the string
```

Searching Strings

The bounded length strings package contains the same search operations we described earlier for fixed length strings. The only difference is the type of the Source string. See that section of this document for examples. As with fixed length strings, there are more than these three search operations in the package:

```ada
function Index (Source  : in Bounded_String;
                Pattern : in String;
                Going   : in Direction              := Forward;

function Index (Source : in Bounded_String;
                Set    : in Maps.Character_Set;
                Test   : in Membership := Inside;
                Going  : in Direction  := Forward)  return Natural;

procedure Find_Token (Source : in Bounded_String;
                      Set    : in Maps.Character_Set;
                      Test   : in Membership;
                      First  : out Positive;
                      Last   : out Natural);
```
Input and Output of Bounded Length Strings

There are no procedures for the input and output of bounded length strings. Instead we use those procedures available for fixed length strings and characters in package Text_IO and conversion functions in the bounded length string package as demonstrated here:

```ada
subtype Name_String is String (1..10);  -- A fixed length string type
Fixed_Name : Name_String;             -- A fixed length string
Length     : Natural;
Name       : Names.Bounded_String;    -- A bounded length string
Ch         : Character;

-- Get a fixed length string and convert it to a bounded length string
Ada.Text_IO.Get_Line (Item => Fixed_Name, Last => Length);
Name := Names.To_Bounded_String (Fixed_Name (1..Length));

-- An alternative method that does not require a fixed length string
Name := Names.Null_Bounded_String;
loop
  exit when Ada.Text_IO.End_Of_Line;
  Ada.Text_IO.Get (Ch);
  Name := Name & Ch;
end loop

-- Display the bounded length string
Ada.Text_IO.Put_Line (Item => Names.To_String (Name));
```

Other Operations

Package Ada.Strings.Bounded contains 66 operations. This document describes only a fraction of them. After reading this far, you should be able to read the specifications of the remaining operations. When you are unsure about some feature in the package, write a test program to clarify it.

**Package Ada.Strings.Unbounded (ARM A.4.5)**

An unbounded string is a varying length string with no upper bound on it length. In practice, the length of an unbounded length string is limited by memory constraints. Package Ada.Strings.Unbounded provides the same operations as found in package Ada.Strings.Bounded.
Number Handling
Ada provides a variety of packages for numerical calculations. These include packages for the elementary functions, complex arithmetic, and random number generation.

Package Ada.Numerics (ARM A.5)

This package is the parent for all other numerics package. It includes two important numeric constants and the exception Argument_Error. This exception is raised by operations in other numerics packages when the actual parameter supplied is outside of the domain that that mathematical operation.

package Ada.Numerics is
    Argument_Error : exception;
    Pi : constant :=  3.14159_26535_89793_23846_26433_83279_50288_41971_69399_37511;
    e  : constant :=  2.71828_18284_59045_23536_02874_71352_66249_77572_47093_69996;
end Ada.Numerics;

Package Ada.Numerics.Generic_Elementary_Functions (ARM A.5.1)

This generic package provides us with the most common mathematical functions. It is a generic package that can be instantiated for any floating point type. Here is an example instantiation.

type High_Precision is digits 18;

package Precise_Functions is new
    Ada.Numerics.Generic_Elementary_Functions (Float_Type => High_Precision);

Here is an example of using the square root function in this package:

    Side_1     : High_Precision;
    Side_2     : High_Precision;
    Hypotenuse : High_Precision;

    Hypotenuse := Precise_Functions.Sqrt (Side_1 ** 2 + Side_2 ** 2);

The details of the domains and ranges of the mathematical functions are described in the ARM. If the cycle is not specified for a trigonometric function, radians are used. If you wish to use degrees, you can either do the conversion yourself (360 degrees = 2\pi radians) or specify a cycle of 360.0.

Package Ada.Numerics.Elementary_Functions (ARM A.5.1)

This package is an instantiation of the generic elementary function package for the predefined type Float.
An Introduction to Ada's Predefined Packages

**Package Ada.Numerics.Float_Random (ARM A.5.2)**

This package provides facilities for the generation of pseudo-random floating point numbers. Here is the part of the specification containing the basic facilities for generating random numbers.

```ada
package Ada.Numerics.Float_Random is

  type Generator is limited private;

  subtype Uniformly_Distributed is Float range 0.0 .. 1.0;

  function Random (Gen : Generator) return Uniformly_Distributed;

  procedure Reset (Gen       : in Generator;
                   Initiator : in Integer);

  procedure Reset (Gen       : in Generator);

end Ada.Numerics.Float_Random;
```

The type Generator allows us to declare as many different random number generators as we desire.

Both Reset procedures are used to initialize a random number generator. The difference between the two is the formal parameter Initiator. The value of Initiator is used to select the first pseudo-random number. The consequences of this is that the same sequence of random numbers is generated each time the program is run. This behavior is very useful in debugging. The Reset procedure without the Initiator parameter uses the system clock to provide the first pseudo-random number. Thus, the sequence of random numbers changes with the time at which the generator is reset.

The following code fragment uses the above package to display 800 random floats between 100.0 and 250.0.

```ada
Random_Value : Float;

-- Initialize the generator from the system clock
Ada.Numerics.Float_Random.Reset (My_Generator);
for I in 1..800 loop
  -- Get a random float between 0.0 and 1.0
  Random_Value := Ada.Numerics.Float_Random.Random (My_Generator);
  -- Scale the float to between 100.0 and 250.0
  Random_Value := 150.0 * Random_Value + 100.0;
  Ada.Float_Text_IO.Put (Random_Value);
end loop;
```
An Introduction to Ada's Predefined Packages


This package provides facilities for the generation of pseudo-random discrete values. Because there are many different kinds of discrete values (integers, Booleans, enumeration types, characters, etc.), this package is a generic package with a single generic formal parameter to specify the particular discrete type you want random values of. Here is the part of the specification containing the basic facilities for generating random discrete values.

```ada
generic
type Result_Subtype is (<>);
package Ada.Numerics.Discrete_Random is

type Generator is limited private;

function Random (Gen : Generator) return Result_Subtype;

procedureReset (Gen : in Generator;
Initiator : in Integer);

procedureReset (Gen : in Generator);

end Ada.Numerics.Discrete_Random;
```

The following code fragment uses the above package to display the values from 100 rolls of a pair of pseudo-dice.

```ada
subtype Die_Range is Integer range 1..6;

-- Create a package for generating random die values
package Die_Random is new Ada.Numerics.Discrete_Random (Die_Range);

Die_1     : Die_Range;
Die_2     : Die_Range;
My_Generator : Die_Random.Generator;

-- Initialize the generator from the system clock
Die_Random.Reset (My_Generator);
for I in 1..100 loop
  -- Roll the dice
  Die_1 := Die_Random.Random (My_Generator);
  Die_2 := Die_Random.Random (My_Generator);
  Ada.Integer_Text_IO.Put (Die_1 + Die_2);
end loop;
```