Introduction to Systematic Programming
Unit 22 - The Standard Ada Libraries

22.1 More on input and output

In this section we will discuss some of the extra facilities provided by the packages `Ada.Text_IO`, `CS_Int_IO` and `CS_Flt_IO`[1]. The list of facilities described below is by no means exhaustive; for a complete list consult the package specification `Ada.Text_IO` in the file `/usr/local/gnatobj/adainclude/a-textio.ads`.

22.1.1 Numerical I/O to/from strings

The package `CS_Flt_IO` provides a third version of `Get` for 'inputting' a floating point value from a character string in memory rather than from an external device such as a terminal or file[2]:

```
Get(From : IN String;  Item : OUT Float;  Last : OUT Positive);
```

This <u>converts</u> a character string supplied as the `From` parameter into the corresponding floating point value which (as usual) it passes back to the caller via the parameter `Item`. The index of the last character of the number 'input' is passed back to the caller via the parameter `Last`. If the string does not contain a valid floating point value, the exception `Data_Error` is raised at run-time.

A third version of `Put` is provided for 'outputting' a floating point value to a character string in memory rather than to an external device such as a terminal or file:

```
Put(To : OUT String;     Item : IN Float;
    Aft : IN Field := 2; Exp  : IN Field := 0);
```

This <u>converts</u> a floating point value supplied as the `Item` parameter into the corresponding character string which it passes back to the caller via the parameter `To`. The formatting of the character string representation of the number is controlled by the parameters `Aft` and `Exp` in the usual way. Note there is no `Fore` parameter; the number of characters before the decimal point is calculated from the size of the string parameter `To`. If the converted number can not be fitted into the string `To`, the exception `Layout_Error` is raised at run-time.

The package `CS_Int_IO` provides similar versions of `Get` and `Put` for 'I/O' of a integer values to/from a character string in memory:

```
Get(From : IN String;  Item : OUT Integer;  Last : OUT Positive);
Put(To : OUT String;  Item : IN Integer;  Base : IN Positive := 10);
```

If the string `From` supplied to this version of `Get` does not contain the representation of a valid integer, the exception `Data_Error` is raised at run-time. Note there is no `Width` parameter for `Put`; instead instead the width is taken as the size of the string AP supplied as the parameter `To`. If the AP string supplied as the parameter `To` is too small to hold the converted number, the exception `Layout_Error` is raised at run-time.

These string 'I/O' procedures are useful for handling 'fancy' input or output of numbers. For example a program which prints monetary values on cheques might output a floating point number with leading asterisks instead of blanks (to make it more difficult for a fraudster to insert extra digits in the amount!):

```
Payment : Float;
Amount : String(1 .. 10);
Blank :  CONSTANT Character := ' ';
......
-- Convert Payment to a string using string-based Put from CS_Flt_IO
Put(To => Amount, Item => Payment);

-- Replace blanks by asterisks
FOR I IN Amount'Range LOOP
   IF Amount(I) = Blank LOOP
      Amount(I) := '*';
   END IF;
END LOOP;

Put(Amount); -- Output the string using Put from Ada.Text_IO
```

---

[1] and the standard packages for numerical I/O: `Ada.Integer_Text_IO` and `Ada.Float_Text_IO`.

[2] The procedures do not perform I/O as such; instead they convert a string value into a numerical value or vice-versa.

In a similar way numerical values separated by asterisks (or any other character which is not valid as part of a number) could be input in two stages: first input the data as a string using `Get` or `Get_Line` from `Ada.Text_IO`, replace the asterisks in the string by blanks and then use the string-based `Get` to convert the string to the required number(s). For example, suppose lines in a data file each consist of two floating point values padded with asterisks rather than blanks; then we may input the values as follows:

```
First, Second : Float;
Line : String(1 .. 80);
Length, Position : Natural;
......
-- input the whole line
Get_Line(Item => Line, Last => Length);

-- replace asterisks by blanks
FOR I IN 1 .. Length LOOP
   IF Line(I) = '*' THEN
      Line(I) := Blank;
   END IF;
END LOOP;

-- Input first number
Get(From => Line(1 .. Length), Item => First, Last => Position);
-- Input second number (starting from where the previous Get left off)
Get(From => Line(Position+1 .. Length), Item => Second, Last => Position);
```

Note the use of slices as the `From` parameter and, in particular, the use of the variable `Position` to record how much of the `Line` has been processed by the first call to the string-based `Get`.

22.1.2 Inputting single characters

Normally input from a terminal keyboard is **line-buffered**, that is the characters typed are not available to a program until the user presses the Return key. Sometimes it is appropriate that characters are available as soon as they are typed and the package `Ada.Text_IO` provides the overloaded procedure `Get_Immediate` for this purpose:

```
Get_Immediate(Item : OUT Character);
Get_Immediate(File : IN File_Type; Item : OUT Character);
```

The first version of `Get_Immediate` inputs a single character from the default input as soon as the user types it. The second inputs the character from the source associated with the parameter `File` (of the type `File_Type` exported by `Ada.Text_IO` as discussed in Unit 20). The use of `Get_Immediate` is only really appropriate when input is being taken from a terminal; when input is being taken from a file on disk `Get_Immediate` behaves exactly like `Get`. A typical application is the following:

```
Reply : Character;
......
OpenInput;   -- redirect input from a file
OpenOutput;  -- redirect output to a file
..........   -- process some data using files
..........
-- now prompt user on terminal whether processing should continue
Put(File => Standard_Output, Item => "Continue? (Y/N) ");
-- get the user's response
Get_Immediate(File => Standard_Input, Item => Reply);
IF Reply = 'Y' OR Reply = 'y' THEN
   .......  -- do some more processing of the data
   .......
END IF;
```

The above two versions of `Get_Immediate` wait until the user presses a key before returning. Therefore they may cause the program to **block** indefinitely waiting for user input and in some programs this may be undesirable. There are also **non-blocking** versions of `Get_Immediate` which take an extra `OUT` mode parameter `Available` of type `Boolean`. If the user has already typed a character, `Available` is set to `True` and the character typed is returned in the normal way via the `Item` parameter. However if the user hasn't typed a character, `Get_Immediate` sets `Available` to `False` and returns immediately and the value of `Item` is undefined. It might be used to poll the keyboard periodically as follows:

```
Char : Character;
CharTyped : Boolean;
.......
Put("Please type a character ");
LOOP
   Get_Immediate(Item => Char, Available => CharTyped);
   IF CharTyped THEN
      ......   -- process the character typed
      ......
      EXIT;    -- then exit the loop
   END IF;
   .......  -- otherwise get on with some other useful work
   .......
END LOOP;   -- now go back to see if the user has typed anything yet
```

22.1.3 The procedure `Look_Ahead`

In many situations it is useful to be able to look at the next character in the input without actually 'consuming' it. For example we might want a procedure to skip over blanks in the input so that that the input position is immediately <u>before</u> the first non-blank character in the input. This is not possible using `Get` in a loop of the form:

```
PROCEDURE SkipBlanks IS
   Char : Character;
BEGIN
   LOOP
      EXIT WHEN End_Of_Line;
      Get(Char);
      EXIT WHEN Char /= Blank;
   END LOOP;
END SkipBlanks;
```

because this 'consumes' the first non-blank character in the input.

The package `Ada.Text_IO` provides the overloaded procedure `Look_Ahead` for use in such circumstances:

```
Look_Ahead(Item : OUT Character; End_Of_Line : OUT Boolean);

Look_Ahead(File : IN File_Type; Item : OUT Character;
           End_Of_Line : OUT Boolean);
```

The first version of `Look_Ahead` inspects the default input and if the current position is at the end of a line it sets the `OUT` parameter `End_Of_Line` to `True`[3], otherwise it sets it to `False` and sets the parameter `Item` to the next character in the default input without actually advancing the input position (and so this character will be available to the next input operation). The second version of `Look_Ahead` works in the same way on the input source associated with the parameter `File`.

Now we can write our `SkipBlanks` procedure properly using `Look_Ahead`:

```
PROCEDURE SkipBlanks IS
   Char : Character;
   EOL : Boolean;
BEGIN
   LOOP
      Look_Ahead(Item => Char, End_Of_Line => EOL);  -- peek ahead
      -- quit when end of line reached or if next char isn't a blank
      EXIT WHEN EOL OR Char /= Blank;
      Get(Char);    -- the next character is a blank, so 'consume' it
   END LOOP;
END SkipBlanks;
```

Using `Look_Ahead` (and `SkipBlanks` above) we can write a neater version of the `GetWord` procedure which appeared in Unit 12:

```
PROCEDURE GetWord(Word : OUT String; Length : OUT Natural) IS
```

[3] The contents of the `OUT` parameter `Item` are undefined when `End_Of_Line` is `True`.

```
      Char : Character;
      EOL : Boolean;
   BEGIN
      Length := 0;
      SkipBlanks;  -- skip to start of the word (or end of line)
      LOOP
         EXIT WHEN Length = Word'Last;  -- exit if Word array is full

         Look_Ahead(Item => Char, End_Of_Line => EOL);  -- peek ahead
         -- quit when end of line reached or if next char is a blank
         EXIT WHEN EOL OR Char = Blank;

         -- input the next character of the word
         Length := Length + 1;
         Get(Word(Length));
      END LOOP;
   END GetWord;
```

Consider now the following problem: a text file contains a unknown number of lines each containing an unknown number of integer data values; data values are separated by one or more blanks. It is required to calculate the average for each line and to count the total number of lines. Providing there are no trailing blanks on any line in the file nor any lines containing only blanks, we can do this quite easily:

```
   Total : Integer;
   Num : Integer;
   Count : Natural;
   NumLines : Natural := 0;
   .......
   WHILE NOT End_Of_File LOOP
      Count := 0;  Total := 0; -- initialise count and total for this line
      WHILE NOT End_Of_Line LOOP
         Get(Num);
         Count := Count + 1;
         Total := Total + Num;
      END LOOP;

      IF Count = 0 THEN
         Put_Line("Empty Line");   -- line contained no data
      ELSE
         -- output the average for the line
         Put(Float(Total)/Float(Count)); New_Line;
      END IF;

      NumLines := NumLines + 1;
      Skip_Line;   -- move to the start of the next line
   END LOOP;

   Put(NumLines);
   Put_Line( lines processed in the file");
```

However this does <u>not work properly if there are trailing blanks on a line</u> because after all numbers on that line have been input, the test `End_Of_Line` will still return `False` and so the inner loop will continue; then the next call to `Get` will move to the next line and input the first data value on that line. Thus the average computed will be for two (or more) lines of the file. If there are trailing blanks on the last line of the file, the situation is worse: after the last data value in the file has been input the inner loop will continue and so the next call to `Get` will attempt to read another integer and so the program will crash with `End_Error` raised. If the procedure `Look_Ahead` were not available it would be quite messy to cater for trailing blanks in the file[4]. However with `Look_Ahead` available the remedy is simple: just call the procedure `SkipBlanks` to consume any troublesome trailing blanks before calling `Get`. The data input loop becomes:

```
   WHILE NOT End_Of_File LOOP
      Count := 0;  Total := 0; -- initialise count and total for this line
```

[4] Try it! One approach would to use `Get_Line` to input a complete line of data into a suitable string variable, remove trailing blanks from the string, then 'input' the data values from the string using the version of `Get` from `CS_Int_IO` discussed in section 22.1.1 above.

```
      SkipBlanks;  -- in case there are completely blank lines in the file
      WHILE NOT End_Of_Line LOOP
         Get(Num);
         Count := Count + 1;
         Total := Total + Num;
         SkipBlanks;  -- deal with possible trailing blanks
      END LOOP;

      IF Count = 0 THEN
         Put_Line("Empty Line");   -- line contained no data
      ELSE
         -- output the average for the line
         Put(Float(Total)/Float(Count)); New_Line;
      END IF;

      NumLines := NumLines + 1;
      Skip_Line;   -- move to the start of the next line
   END LOOP;

   Put(NumLines);
   Put_Line( lines processed in the file");
```

It is perhaps worth emphasizing that it is only trailing blanks on a line that cause potential problems with numerical input; the standard `Get` procedures from `CS_Int_IO` and `CS_Flt_IO` automatically skip any blanks occurring before data values and so are perfectly adequate for many applications.

22.1.4 More on formatted output

So far in these Units we have used output files with variable length lines. If we wish to start a new line in the output we simply call the procedure `New_Line`. However, in some situations it is useful to be able to specify a maximum line length for output so that if data will not fit on the current line, a new line is begun automatically and all the data is output on the new line, otherwise it is output in the normal way. The following subprograms from `Ada.Text_IO` are useful in this context:

```
PROCEDURE Set_Line_Length(To : IN Count);
PROCEDURE Set_Line_Length(File :IN  File_Type; To : IN Count);
FUNCTION Line_Length RETURN Count;
FUNCTION Line_Length(File : File_Type) RETURN Count;
```

where `Count` is a non-negative integer subrange[5] and is exported by `Ada.Text_IO`. A call to the first version of `Set_Line_Length` with a positive AP value `N` supplied for the FP `To` sets the maximum length of lines in the default output to `N` characters; supplying a zero value for the parameter `To` allows lines of unlimited length in the output so that a new line is only started by an explicit call to `New_Line`. `Ada.Text_IO` exports the constant

```
Unbounded : CONSTANT Count := 0;
```

which should be used (rather than the explicit value zero) for reasons of clarity if no maximum line length is required. By default output files have line length set to `Unbounded`.

The second version of `Set_Line_Length` behaves in the same way except that it affects the output file associated with the parameter `File` (as discussed in Unit 20). `Set_Line_Length` is not meaningful for input files and so will not work.

The first version of the function `Line_Length` returns the current maximum line-length set for the default output, or zero if no maximum length has been set (the default). The second version returns the current maximum line-length set for the file associated with the parameter `File` (which may refer to either an input or an output file).

The following subprograms enable the input or output position on a line to be controlled:

```
PROCEDURE Set_Col(To : IN Positive_Count);
PROCEDURE Set_Col(File :IN  File_Type; To : IN Positive_Count);
FUNCTION Col RETURN Count;
FUNCTION Col(File : File_Type) RETURN Positive_Count;
```

[5] the actual range is implementation dependent, but is always large enough for all practical purposes. Also be aware that if you declare a variable `Count` in your program, the type `Count` exported by `Ada.Text_IO` will be hidden (see Unit 21) and so will only be accessible if you use its fully qualified name `Ada.Text_IO.Count`.

where Positive_Count is subrange type exported by Ada.Text_IO which coincides with the subrange Count discussed above except that zero is excluded. Calls to Col return the current position on the line. A call to Set_Col advances the current position on the line to the column specified by the AP value supplied for the FP To. For output files blanks are output to advance the position and for input characters are simply skipped. Set_Col can never go backwards: if the AP value supplied for the FP To is less than the current position, the position is moved to the specified column on the next line of the file by first making an implicit call to New_Line for output or to Skip_Line for input. The versions of the subprograms with no file parameter always act on the default output whereas the versions with a file parameter act on the file associated with the parameter File and can be used with either input or output files[6].

These facilities are useful for producing formatted output such as tables. As a simple example consider the output of three integer values on a line left-justified at columns numbers 20, 40 and 60:

```
A, B, C : Integer;
.....
Set_Col(20); Put(A);
Set_Col(40); Put(B);
Set_Col(60); Put(C);
```

Of course, if right-justified output were required at the same positions was required, we would supply a suitable Width parameter in the calls to Put (as discussed in Unit 4):

```
Put(Item => A, Width => 20);
Put(Item => B, Width => 20);
Put(Item => C, Width => 20);
```

22.2 Character handling

The package Ada.Characters.Handling provides a number of useful functions for testing single characters, for example:

```
FUNCTION Is_Control(Item : IN Character) RETURN Boolean;
```

This function returns True if the character supplied as AP is a control character (that is if it has an internal code in the range 0..31 or 127..159) and False otherwise. The character testing functions (which all have a single IN mode parameter Item of type Character and return a Boolean value) include

Function	tests for	True for char. codes
Is_Control	control character	0..31, 127..159
Is_Graphic	graphic character (i.e. not a control char)	32..126, 160..255
Is_Lower	lower case letter	97..122, 223..246, 248..255
Is_Upper	upper case letters	65..90, 192..214, 216..222
Is_Letter	letter (either upper or lower case)	Is_Upper OR Is_Lower
Is_Digit	a digit '0'..'9'	48..57
Is_Hexadecimal_Digit	'0'..'9', 'A'..'F', 'a'..'f'	48..57, 65..70, 97..102
Is_AlphaNumeric	either a letter or a digit	Is_Letter OR Is_Digit
Is_Special	graphic character but not alphanumeric (i.e. punctuation, symbols or operators)	Is_Graphic AND NOT Is_AlphaNumeric
Is_ISO_646	ASCII character	0..127

Note that the character set used is the full Latin_1 set (ISO 6429)[7] with character codes in the range 0..255. Thus letters include extra letters ß, æ etc. and accented characters é, è, ü, ä,, å, ñ etc. used in the major West European languages. There are also extra arithmetic operators ÷, □, etc., symbols £, ¥ etc. and foreign punctuation signs ¿, ¡ etc.. The basic ASCII character set (ISO 646) is the subset of Latin_1 with character codes in the range 0..127.

---

6 if the current column position of the default input needs to be manipulated then the AP Current_Input (see Unit 20) should be used as the File parameter with these subprograms.

7 ISO stands for International Standards Organisation.

There are a number of functions for converting single characters and strings to upper or lower case:

```
FUNCTION To_Lower(Item : IN Character) RETURN Character;
FUNCTION To_Upper(Item : IN Character) RETURN Character;
FUNCTION To_Lower(Item : IN String) RETURN String;
FUNCTION To_Upper(Item : IN String) RETURN String;
```

Only letters (including foreign and accented characters) are altered; other characters are unchanged. These functions do not, of course, alter the parameter `Item`, they copy the parameter and return the converted copy.

Not all hardware (printers or terminals) can handle the full Latin_1 character set properly and so a subtype `ISO_646` consisting of only ASCII characters (codes 0..127) is defined and exported by `Ada.Characters.Handling`. There is predicate function `Is_ISO_646` for testing whether a string is composed only of characters belonging to the basic ASCII (ISO 646) set:

```
FUNCTION Is_ISO_646(Item : IN String) RETURN Boolean;
```

Two functions are provided for replacing non-ASCII characters by a substitute (a space is the default) from the ASCII character set; one version converts a single character and the other a whole string:

```
FUNCTION To_ISO_646(Item : IN Character;
                    Substitute : IN ISO_646 := ' ') RETURN ISO_646;

FUNCTION To_ISO_646(Item  : IN String;
                    Substitute : ISO_646 := ' ') RETURN String;
```

The package `Ada.Characters.Latin_1` defines named constants for all the characters in the Latin_1 set (except upper case letters and digits). These enable Ada programs handling the full Latin_1 set to be developed on systems where the hardware does not support the full character set. The constants include `Space`, `Comma`, `Full_Stop`, `Semicolon`, `Colon`, `Hyphen`, `Pound_Sign`, `Yen_Sign`, `Fraction_One_Half`, `Superscript_Two`, `Multiplication_Sign`, `Division_Sign`, `LC_E_Acute`, `UC_E_Acute` etc. all with fairly obvious meanings. For a complete list consult the package specification `Ada.Characters.Latin_1` which on Computer Science UNIX systems at Aston may be found in the file `/usr/local/gnatobj/adainclude/a-chlat1.ads`.

22.3 String handling

Strings can be joined with the concatenation operator `&`, compared with the standard comparison operators =, >, < etc. and, by using slices and assignment, we may select or alter sections of strings, for example:

```
Word1 : String :="Hello";
Word2 : String :="there";
Phrase : String(1 ..11);
.........
Phrase := Word1 & ' ' & Word2;  -- sets Phrase to "Hello there"
Word2(1..3) := "Cla";           -- changes Word2 to "Clare"
Put(Word1(1..4));               -- output "Hell"
```

Although these operations are adequate for many string operations, we must be careful that the lengths of the strings on the left and right hand sides of an assignment are always the same length, otherwise an error will occur:

```
Phrase := Word1 & " Les";   -- !!?? illegal as rhs has length 9
```

The package `Ada.Strings.Fixed` provides a number of more sophisticated string handling facilities. For example the procedure `Move` allows copying when the source and destination strings are of different lengths. For example

```
Move(Source => Word1 & " Les", Target => Phrase);
```

copies the string "Hello Les" into the string variable `Phrase` and automatically pads out the string to the correct length by adding two extra `Space` characters at the end to give `"Hello Les∆∆"`. By default the character used to pad out the target string is a space and the padding characters are added at the end. If the `Source` string (after trimming off any padding spaces from both ends) is longer than the `Target` string then an error occurs at run time (the exception `Length_Error` exported by the package `Ada.Strings` is raised). However this behaviour can be altered by supplying extra parameters in the call to `Move` whose full declaration is

```
PROCEDURE Move(Source : IN String; Target : OUT String;
               Drop : IN Truncation := Error;
               Justify : IN Alignment := Left;
               Pad : IN Character := Space);
```

The parameter `Justify` can have one of the values `Left`, `Right` or `Center`[8] and causes any necessary padding characters to be added at the end, the beginning or equally at both ends[9] of the string respectively. The `Pad` parameter can be used to specify that a character other than a space is to be used for padding. The parameter `Drop` controls what happens if the `Source` string (after trimming off padding characters) is longer than the `Target`: it can have one of the values `Left`, `Right` or `Error`; setting this parameter to `Left` causes enough non-padding characters to be discarded from the beginning of the source string so that it fits into the target string, for the value `Right`, characters are dropped from the end. If the value is `Error`, `Length_Error` is raised at run-time if non-padding characters would need to be dropped. Thus the calls

```
Move(Source => Word1 & " Les", Target => Phrase, Justify => Right);
Move(Source => Word1 & " Les", Target => Phrase); Justify => Center);
Move(Source => Word1 & " Elizabeth", Target => Phrase, Drop => Right);
```

assign the strings `"ΔΔHello Les"`, `"ΔHello LesΔ"` and `"Hello Eliza"` respectively to the string variable `Phrase`.

The enumeration types `Truncation and Alignment` appearing in the heading of `Move` are exported by the package `Ada.Strings` along with several other enumeration types; these are used in a other subprograms exported by `Ada.Strings.Fixed`. These are defined as[10]

```
TYPE Alignment  IS (Left, Right, Center);
TYPE Truncation IS (Left, Right, Error);
TYPE Direction  IS (Forward, Backward);
TYPE Trim_End   IS (Left, Right, Both);
```

Thus programs which use the string library will normally need to import both `Ada.Strings` and `Ada.Strings.Fixed`:

```
WITH Ada.Strings;       USE Ada.Strings;
WITH Ada.Strings.Fixed; USE Ada.Strings.Fixed;
```

The function `Index` allows a `Source` string to be searched for a `Pattern` sub-string; the direction of search can either be `Forward` from the beginning of the string (the default) or `Backward` from the end. It returns the index of the start of the first occurrence (in the specified `Direction`) of the sub-string `Pattern` in the `Source` string. If the `Pattern` is not present in the `Source` string, zero is returned. Another useful function `Index_Non_Blank` searches a string and returns the index of the first non-blank character (in the specified `Direction`) in the string. A third function `Count` is provided which counts the number of occurrences of a `Pattern` string in a `Source` string. Here are a few examples of their use:

```
Name : String := "Barbara Barnes";
I : Natural;
......
I := Index(Source=>Name, Pattern =>"bar"); -- sets I to 4
I := Index(Source=>Name, Pattern=>"barnes"); -- search fails so sets I to 0
I := Index(Source=>Name, Pattern=>"Bar", Going=>Forward); -- sets I to 1
I := Index(Source=>Name, Pattern=>"Bar", Going=>Backward); -- sets I to 9
I := Index_Non_Blank(Source=>Name, Going=>Backward); -- sets I to 14
I := Count(Source=>Name, Pattern=>"Bar");  -- sets I to 2
```

There are a number of useful functions for editing strings in various ways:

```
FUNCTION Trim(Source : String; Side : Trim_End) RETURN String;
```

`Trim` returns a modified copy of the `Source` string with all spaces removed from the start, from the end, or from both ends of the string when `Side` is `Left`, `Right` or `Both` respectively.

---

8 so that the copy of the `Source` string is left-justified, right-justified or centred in the `Target` string respectively.

9 if an odd number of pad characters are added, the extra one is added at the end.

10 Note the overloading (see Unit 21) of the enumeration literals `Left` and `Right` here!

```
FUNCTION Insert(Source : String;
                Before : Positive; New_Item : String) RETURN String;
```

`Insert` copies of the `Source` string and then inserts the string `New_Item` into the copy just before the character with index `Before`. The modified string is returned as the value of the function.

```
FUNCTION Overwrite(Source : String; Position : Positive;
                   New_Item : String) RETURN String;
```

`Overwrite` copies of the `Source` string and then overwrites a portion of the copy starting at the character with index `Position` with the string `New_Item` . The modified string is returned as the value of the function.

```
FUNCTION Delete(Source : String;
                From : Positive; Through : Natural) RETURN String;
```

`Delete` copies of the `Source` string and then deletes the characters with indices between `From` and `Through` (inclusive) from the copy. The modified string is returned as the value of the function. If `Through < From`, no characters are deleted from the returned string.

```
FUNCTION Replace_Slice(Source : String; Low : Positive;
                       High : Natural; By : String) RETURN String;
```

`Replace_Slice` copies of the `Source` string and then replaces the characters with indices between `Low` and `High` (inclusive) from the copy with the string `By`. The modified string is returned as the value of the function. The slice replaced may have a different length from the string `By`. If `High < Low`, no characters are replaced, but instead the `By` string is inserted just before the character with index `Low`.

If the return value of any of the above five functions is assigned to a string variable some care must be exercised to ensure that the target string is of the correct size, otherwise a `Constraint_Error` will be raised at run time. However, sometimes the returned string can be used directly without its size being known, for example as a parameter in another subprogram or as the initialiser in the declaration of a string variable. Suppose `S` is a string variable with the value `"ΔΔΔHello WorldΔΔΔΔ"`:

```
-- The size of T is deduced from the size of the return value of trim
-- i.e. 11 and T is initialised to "Hello World".
T : String := Trim(Source => S, Side => Both);

-- Output S with the 4 trailing blanks removed and start a new line.
Put_Line(Trim(Source => S, Side => Right));

-- Output "Good-bye World"
Put(Replace_Slice(T, 1, 5, "Good-bye"));
```

There are procedure versions of `Trim`, `Insert`, `Overwrite`, `Delete` and `Replace_Slice` that modify the string parameter `Source` which now has mode `IN OUT`. `Insert` increases the length of the string and so takes an optional fourth parameter `Drop` of type `Truncation` (as in `Move`). If the modified string, after removing any padding characters from its ends, is still too long to fit, non-padding characters are truncated from the front of the string until it fits when `Drop` is `Left`, or from the end of the string when `Drop` is `Right`. Alternatively, when `Drop` is `Error` (the default), `Length_Error` is raised in these circumstances. Similarly `Overwrite` usually increases the length of the string and takes `Drop` as an optional extra parameter, but in this case its default value is `Right` so that, by default, non-padding characters are dropped from the end of the modified string if necessary.

The procedures `Delete` and `Trim` usually shorten the string and take optional parameters, `Justify` and `Pad` to control the justification and the character used to pad the modified string (with the same meaning and default values as in `Move`). Although it might seem paradoxical that the procedure version of `Trim` removes padding only to replace it, it can be used usefully (for example) to move trailing blanks to the front of the string in a call such as

```
Trim(Source => S, Side => Right, Justify => Right);
```

The procedure `Replace_Slice` can either increase or decrease the length of the string and takes three extra optional parameters `Drop`, `Justify` and `Pad` (with the same meaning and defaults as in `Move`). Thus, assuming `S` is a string variable with the value `"ΔΔΔHello WorldΔΔΔΔ"`, we could set `S` to `"Good-bye WorldΔΔΔΔ"` by doing (since `Justify => Left` is the default):

```
Replace_Slice(S, 4, 9, "Good-bye");
```


Finally the package `Ada.Strings.Fixed` provides extra overloadings of the multiplication operator `"*"` which allow a character or a string to be replicated a specified number of times. For example:

```
-- declare T to be of length 80 with alternate minus and plus signs
T : String := 40 * "-+";

Put_Line(72 * '/');  -- output a line of 72 slashes
```

There are many more advanced string searching, translating and editing facilities available and also a package for defining and manipulating variable length strings, but discussion of these is beyond on the scope of this course (see the package specifications of `Ada.Strings.Fixed`, `Ada.Strings.Maps`, `Ada.Strings.Bounded` and `Ada.Strings.Unbounded` for details).

22.4 Getting More Information on the standard Ada Library Packages

The package specifications of all the standard Ada packages in the Gnat library may be found in the directory

```
/usr/local/gnat3.13p/lib/gcc-lib/sparc-sun-solaris2.5.1/2.8.1/adainclude
```

The UNIX filenames of each package specification can be found by consulting the file

```
/usr/local/staffstore/CSAdaLib/gnat_filenames.txt
```

For complete library documentation, consult the Predefined Language Environment section of the Ada Reference Manual available on-line at the URL:

```
http://www.adahome.com/rm95/
```