Introduction to Systematic Programming
Unit 15 - More on Arrays and Strings.

15.1 Unconstrained array types

In Units 10 and 11 we saw how to declare array types, for example:

```
NumHts : CONSTANT Integer := 100;
NumWts : CONSTANT Integer := 200;
TYPE HeightReadings IS ARRAY (1 .. NumHts) OF Float;
TYPE WeightReadings IS ARRAY (1 .. NumWts) OF Float;
```

The array types defined by this sort of definition are known as **constrained array types** since the size of the array and the index bounds are fixed (or constrained) at the point of the type declaration. Such array types may be used in much the same way as the pre-defined types to declare global and local variables and in the formal parameter specifications in procedure or function headings. However arrays declared in this way have a number of limitations which will now be considered.

Suppose we define a function `Sum` to add up and return the sum of all the values in an array of type `HeightReadings`:

```
FUNCTION Sum (Height : HeightReadings) RETURN Float IS
   TotalSoFar : Float := 0.0;
BEGIN
   FOR I IN 1 .. NumHts LOOP
      TotalSoFar := TotalSoFar + Height(I);
   END LOOP;
   RETURN TotalSoFar;
END Sum;
```

This function could then be called to determine the sum of *any* array of type `HeightReadings`. However, this procedure is not very flexible as it may only be used with arrays of the type `HeightReadings`. We cannot use it to add up the elements of an array of `Float` values of any other type (such as the type `WeightReadings`) which has a different index range[1]. We would need to define a different (but very similar) version of the `Sum` function for each and every array type. It is clearly inconvenient to write large numbers of almost identical procedures and functions; what is needed is the possibility of writing procedures or functions which could process arrays of a given element type and given index type, but of arbitrary size. In order to be able to do this we need to discuss **unconstrained array types**, that is array types where the index bounds are not given at the point where the type is defined.

For example, consider the declarations:

```
TYPE Vector IS ARRAY (Integer RANGE <>) OF Float;
SUBTYPE Vector6 IS Vector(1 .. 6);
SUBTYPE BigVector IS Vector(1 .. 100);
```

This declares `Vector` as the name of an array type with elements of type `Float` and with an index of type `Integer`. However the lower and upper bounds of the index (and hence the size of the array) are not given at this stage and the notation `RANGE <>` is meant to convey that this information is to be "filled-in later". Once an unconstrained array type has been declared, various subtypes (such as `Vector6` and `BigVector`) may be declared by supplying **index constraints** (that is by specifying the lower and upper bounds of the index).

Array variables of these subtypes may then be declared in the normal way:

```
U, V : Vector6;
X, Y : BigVector;
```

[1] In fact, even if the index ranges are the same (ie. if `NumHts = NumWts`), the types `HeightReadings` and `WeightReadings` are rightly regarded as different types in Ada (as they have different type names). Thus it is still the case that the function `Sum` cannot be used to add up the elements of an array of type `WeightReadings`.

Alternatively one may declare array variables directly by attaching index constraints to the type name:

```
Z : Vector(0 .. 5);
W : Vector(1 .. 6);
```

The variables `Z` and `W` are said to belong to **anonymous subtypes** (anonymous - since they are not of a named subtype) of the type `Vector`. With either method of declaration the index bounds (and hence the size of the array) are known when an array variable is declared and so the compiler is able to allocate the appropriate amount of storage for the array. Note, however, that it illegal to attempt to declare variables using an unconstrained array type directly:

```
          T : Vector;        -- !? Illegal in Ada
```

Since the size of such an array is not specified, how could the compiler possibly allocate storage for it? Note also that the declaration of the unconstrained array *type* itself does not cause the same problem since type declarations do not cause storage to be allocated.

Now we may define a function with a formal parameter of the unconstrained array type `Vector`:

```
FUNCTION Sum (A : Vector) RETURN Float IS
   TotalSoFar : Float := 0.0;
BEGIN
   FOR I IN A'Range LOOP
      TotalSoFar := TotalSoFar + A(I);
   END LOOP;
   RETURN TotalSoFar;
END Sum;
```

The use of `A'Range` in the `FOR` loop will be explained in [15.2] below. The function `Sum` may now be used to add up the elements of *any* array of type `Vector` (no matter what its size or index bounds are) by writing call steps of the form:

```
      Total := Sum(U);  -- Set Total to the sum of the 6 elements of U
```

or
```
      Total := Sum(X)   -- Set Total to the sum of the 100 elements of X
```

or
```
      Put(Sum(Z));      -- Output the sum of the 6 elements of Z
```

Clearly this is a great improvement as the function `Sum` can be used to process arrays with arbitrary index bounds provided only that the array is of the type `Vector`. However it should be emphasised that `Sum` could still not be used to add up the elements of an array of any other type. For example, it would be illegal to call `Sum` with an AP of type `HeightReadings` or `WeightReadings`.

The general form of an unconstrained array type declaration is:

`TYPE` *`Type_name`* `IS ARRAY(`*`Index_type_name`* `RANGE <>) OF` *`Element_type_name`*`;`

where *`Type_name`* is the name of the unconstrained array type being declared,
`Index_type_name` is the *name* of the type or subtype of the array index , and
`Element_type_name` is the *name* of the type or subtype of the array elements.

Here are some example declarations of unconstrained array types, subtypes and array variables:

```
TYPE DayOfWeek IS (Mon, Tues, Wed, Thurs, Fri, Sat, Sun);
TYPE DayArray  IS ARRAY (DayOfWeek RANGE <>) OF Float;
SUBTYPE WeekdayArray IS DayArray(Mon .. Fri);
HrsWorked : WeekdayArray;

TYPE IntArray  IS ARRAY (Natural RANGE <>) OF Integer;
SUBTYPE AgeCountArray IS IntArray(0 .. 120);
AgeCounts : AgeCountArray;

TYPE CharCountArray IS ARRAY (Character RANGE <>) OF Natural;
CapitalLetterCounts : CharCountArray('A' .. 'Z');
```

Note that unconstrained array types are, by their very nature, general (being of any size) and so will tend to be given fairly non-specific names such as `Vector`, `IntArray` and `DayArray`. However when declaring subtypes or array variables of these types we should normally choose more specific and meaningful names related to the problem domain under consideration.

Note also that it is only the index range that may be constrained when introducing a new array subtype. The element type (or subtype) is completely fixed in the declaration of the unconstrained array type. For example, it is not possible in Ada to define an array type whose elements are of type `Integer` and later constrain the element type to be a subtype of `Integer`.

15.2 Array attributes

When a procedure or function (such as `Sum` on page 2 of this unit) with a formal parameter `A` (say) of an unconstrained array type is written, the index range of any actual parameter to be supplied in a call step is not known and, indeed, this range may be different each time the procedure is called. However, inside the procedure or function body it is clearly necessary to know the index range in order to be able to refer to the elements of the array. In Ada this information is obtained by calling the four **array attribute functions**:

`A'First`	gives the lower bound of the index of the array `A`
`A'Last`	gives the upper bound of the index of the array `A`
`A'Length`	gives the number of elements of the array `A`
`A'Range`	gives the index range of the array `A` (ie. `A'First .. A'Last`)

As a procedure (or function) is called the attributes of any array APs are (in effect) passed into the procedure and become the attributes of the corresponding FPs. Then the appropriate amount of storage is allocated for the array FPs. Next the elements of any array FPs of mode `IN` and `IN OUT` are initialised by (in effect) copying the elements of the corresponding array APs. Finally when the procedure 'returns', the elements of any array FPs of mode `OUT` or `IN OUT` are (in effect) copied out to the corresponding array APs.

The array attributes are often used for the control of loops which are to process array FPs. For example, while the function `Sum` defined on page 2 of this unit is executing, the attribute `A'Range` gives the index range of the array AP used in the call step which invoked the function. This range may then be used to control the `FOR` loop in the function and so enable it to add up all the elements of the array AP.

Although the most frequent use of array attribute functions is with formal array parameters in procedures and function bodies, these attribute functions may also be used directly with array *types* or *subtypes*. For example to add corresponding elements of certain arrays `A` and `B` and assign the results to the elements of an array `C` we might write:

```
TYPE IntArray IS ARRAY (Integer RANGE <>) OF Integer;
SUBTYPE Array10 IS IntArray(1 .. 10);
A, B, C : Array10;
.................
FOR I IN Array10'Range LOOP
   C(I) := A(I) + B(I);
END LOOP;
```

Here the use of the attribute attached to the subtype name `Array10` (rather than to one of the array variables `A`, `B` or `C`) preserves the 'symmetry' of the program code and helps to indicate that this attribute applies to all three array variables.

As one might expect, using an attribute function with an unconstrained type (for example `IntArray'Range`) produces a compilation error since no index constraints have been specified.

15.3 More on array assignment

In Unit 11 we saw that whole array assignment of one array to another was permitted provided that the arrays were of identical types (and so necessarily had the same element type, index type and index range). Whole array assignment is also possible in more general circumstances for arrays which are subtypes of the *same* unconstrained type. For example given the declarations in [15.1] above it should come as no surprise that the following assignments are both valid:

```
U := V;       -- Both arrays of subtype Vector6
X := Y;       -- Both arrays of subtype BigVector
```

However, it is not necessary for the subtypes to be identical; it suffices that the two subtypes have the same number of elements (but not necessarily the same index ranges). For example both the following assignments are also valid:

```
            U := Z;     -- Both arrays of type Vector each with 6 elements
            Z := W;
```

In the first assignment the index range of `Z` is `0..5` and that of `U` is `1..6` and the elements of `Z` are 'slid' one position 'upwards' so that they 'fit' into `U`, that is:

```
      U(1) := Z(0),  U(2) := Z(1), ...,  U(5) := Z(4),  U(6) := Z(5)
```

Similarly in the second assignment the elements of `W` (index range `1..6`) are 'slid' one position 'downwards' to fit into `Z` (index range `0..5`).

However for arrays of two subtypes (of the same unconstrained type) with different numbers of elements, a run-time error will occur if we attempt to use whole array assignment, for example:

```
            X := V;     -- !? Illegal in Ada   (arrays of different sizes)
            U := Y;     -- !? Illegal in Ada   (arrays of different sizes)
```

15.4 Testing whether two arrays are equal

The standard comparison operators = and /= may be used to test whether two arrays of the *same type* are equal or not. The condition `U = V` (where both arrays are of subtype `Vector6`) yields the value `True` only when corresponding elements of `U` and `V` are all equal.

When two arrays of different subtypes of an unconstrained array type are compared, 'sliding' takes place (as with assignment) before the elements are compared, for example the comparison `Z = U` yields the value `True` only if:

```
      Z(0) = U(1),  Z(1) = U(2),  ...,  Z(4) = U(5),  Z(5) = U(6)
```

It is not an error to compare two arrays of the same type but of different sizes; in Ada they are always regarded as not equal.

15.5 Array slices

In Ada it is possible to manipulate a **slice** of an array (that is a series of consecutive elements of the array) as a whole. For example we could assign the six elements of the array `U` (of subtype `Vector6`) to the first six elements of the array `X` (of subtype `BigVector`) as follows:

```
                  X(1 .. 6) := U;
```

The remaining 94 elements of `X` (indexed `7..100`) would be unaffected by this assignment. Similarly the assignment:

```
                  Z := X(90 .. 95);
```

'slides' the 6 elements of the slice of `X` to 'fit' into the array `Z` (index range `0..5`) and thus has the same effect as the six separate assignments:

```
                  Z(0) := X(90);   Z(1) := X(91);  Z(2) := X(92);
                  Z(3) := X(93);   Z(4) := X(94);  Z(5) := X(95);
```

The general form of an array slice is *`Array_name(Index_range)`* where *`Index_range`* may be an explicit range (as above) or a subtype name. Slices may appear virtually anywhere in a program that an array (of the same type and size) would be valid.

15.6 Constant arrays

It is possible to define **constant arrays** of an array type; the elements of such an array are given values when it is declared using appropriate array aggregates. The values of the array elements cannot not be altered by subsequent program steps. For example:

```
TYPE MonthOfYear IS (Jan, Feb, Mar, Apr, May, Jun,
                     Jul, Aug, Sep, Oct, Nov, Dec);
TYPE MonthCounts IS ARRAY (MonthOfYear) OF Natural;
DaysIn : CONSTANT MonthCounts
                      := (Jan|Mar|May|Jul|Aug|Oct|Dec => 31,
                          Apr|Jun|Sep|Nov => 30, Feb => 28);
```

Array constants of this sort are quite useful in practice and can often avoid the need to use a `CASE` step. For example we may write (assuming `ThisMonth` and `MonthLength` are a variables of type `MonthOfYear` and `Integer` respectively):

```
MonthLength := DaysIn(ThisMonth);
```

rather than:

```
CASE ThisMonth IS
   WHEN Jan|Mar|May|Jul|Aug|Oct|Dec => MonthLength := 31;
   WHEN Apr|Jun|Sep|Nov             => MonthLength := 30;
   WHEN Feb                         => MonthLength := 28;
END CASE;
```

15.7 The type `String`

Character strings (that is arrays of element type `Character`) are frequently used in programming. Ada provides a *pre-defined* unconstrained array type `String` of the form:

```
TYPE String IS ARRAY (Positive RANGE <>) OF Character;
```

We may declare subtypes and array variables of the type `String` in the standard way:

```
LineLength : CONSTANT Integer := 80;
SUBTYPE LineType IS String(1 .. LineLength);
Line : LineType;

MaxIDLength : CONSTANT Integer := 8;
SUBTYPE UnixUserID IS String(1 .. MaxIDLength);
CurrentUser : UnixUserID;
```

All the features of Ada available for use with array types are naturally available for the type `String`. For example, suppose we wish to declare a function to convert any lower-case characters in an array of type `String` to upper-case and return the modified string as the value of the function. We could write (using the function `UpperCase` from [13.4] to convert individual characters of the string to upper-case):

```
FUNCTION UpperCaseString (OldStr : String) RETURN String IS
   NewStr : String(OldStr'Range);
BEGIN
   FOR I IN OldStr'Range LOOP
      NewStr(I) := UpperCase(OldStr(I));
   END LOOP;
   RETURN NewStr;
END UpperCaseString;
```

The above example also illustrates that it is possible to return an array as the value of a function. Having defined our function we could perhaps use it to convert the contents of the string `Line` to upper-case as follows:

```
Line := UpperCaseString(Line);
```

In addition to the standard array features Ada provides a number of additional features which facilitate string processing.

15.7.1 String literals

String literals are constant arrays of the type `String` and are written enclosed in double quotes. We have been using string literals since Unit 2 in output steps such as:

```
Put("The average is ");
```

Note that in string (and character) literals the case of characters is significant unlike in the rest of Ada; thus, for example, `"ALAN"` and `"Alan"` are regarded as different strings. String literals are, in fact, just a special shorthand notation for array aggregates with elements of type `Character`; for example:

`"Les"` is equivalent to the aggregate `(1=>'L', 2=>'e', 3=>'s')` but the former is obviously much more convenient. String literals are often used in declarations of string constants, for example:

```
      ErrMsg : CONSTANT String := "***** Error in data *****";
```

Here `ErrMsg` is declared as an array constant of subtype `String(1 .. 25)`; note that the index constraints are deduced by the Ada compiler from the length of the string literal and so need not be specified explicitly. Subsequently we may write:

```
Put(ErrMsg);
```

rather than the more long-winded: `Put("***** Error in data *****");`

Similarly we may use string literals in initialised variable declarations, for example:

```
CurrentUser : UnixUserID := "hazlewlj";
SuperUser   : UnixUserID := "root    ";
```

However, note that if the literal string is shorter than `MaxIDLength` (ie. `8`) characters, we must be careful to pad it with the appropriate number of blanks (so that the arrays are of the same size), otherwise an error will occur when the whole array assignment is performed.

15.7.2 String slices

The flexibility of string handling in Ada is enhanced by using slices. They enable us to manipulate continuous **substrings** as a whole simply by specifying the index bounds of the required slice. For example the assignment:

```
CurrentUser(1 .. 7) := "barnesa";
```

stores the characters `'b'`, `'a'`, `'r'` etc. in the first 7 elements of the string `CurrentUser`, but does not alter the 8th character of this string which (given the declaration above) would still be `'j'`. If we use this method of assigning values to string variables we must keep track of the length of the current string stored in the variable by some means or other, perhaps by storing it in a variable `CurrentIDLength` (say) of the `Integer` subtype `Natural`.

We may even use slices in the following way:

```
CurrentUser(4 .. 6) :=  CurrentUser(1 .. 3);
```

which would change the first 7 characters of `CurrentUser` to `"barbara"`!

15.7.3 String concatenation

The **concatenation** (or **catenation**) operator `&` may be used to join two arrays of the type `String` end to end (or more generally two arrays of the same unconstrained type). For example:

`"Hello " & "World"` produces the string `"Hello World"`

and:

```
Put("No mail for " & CurrentUser(1 .. 7));
```

produces the output (assuming the above declarations and assignments):

```
No mail for barbara
```

The concatenation operator may also be used to append a single character to either end of a string.

15.7.4 String comparison operators

The relational operators >, <, <= and >= (as well as = and /=) may be used to compare two arrays of type `String`[2]. The order is lexicographic based on the ordering of characters determined by their Latin-1 (ISO Standard 8859-1) codes (for strings only involving letters this coincides with standard dictionary order); for example:

`"pen" > "sword"`	yields the value	`False`	(as `'p'<'s'`)
`"bag" < "bat"`	yields the value	`True`	(as `'g'<'t'`)
`"mat" < "matt"`	yields the value	`True`	(as the 2nd string is longer)
`"pen" > "Sword"`	yields the value	`True`	(as `'p'>'S'`)
`"100" < "20"`	yields the value	`True`	(as `'1'<'2'`)

[2] More generally they may be used to compare two arrays of an array type whose elements are of any *discrete* type.

15.7.5 String input and output

The library package `Ada.Text_IO` contains versions of the procedures `Get` and `Put` for the input and output of strings. The procedure `Put` has been used extensively in these units already and requires no further comment except to emphasize that it may be used to output the contents of string variables and string slices as well as string literals.

The procedure `Get` takes a FP `Item` of the unconstrained type `String` and is used to input character strings and store them in the array supplied as the AP in procedure calls. The AP must, of course, be an array variable (or slice) of some constrained subtype of the type `String`. For example:

```
        Get(Item => CurrentUser);  -- or simply Get(CurrentUser);
```

The behaviour of `Get` needs a little explanation: firstly it skips over any leading e-o-l markers until it finds a regular character, it then 'reads' characters in and stores them in the string variable until the string is *completely full*. If it encounters further e-o-l markers before the string is full, it simply skips these and continues reading characters from the next line of input. Thus the above `Get` call step would 'consume' exactly 8 characters including blanks but excluding e-o-l markers.

A common error when using `Get` for interactive input is for the user to type an insufficient number of characters to fill the string and to press the `Return` key. The program will appear to 'hang' but in reality it is simply waiting for the user to type more characters. Also when using `Get` for file input one must be careful to ensure that there are sufficient characters 'left to read' in the file in order to fill the string before end of file is reached, otherwise a run-time error will occur as `Get` attempts to read past the end of file.

The library package `Ada.Text_IO` contains two other procedures `Get_Line` and `Put_Line` for the input and output of strings[3]. A call to the procedure `Put_Line` is simply equivalent to the corresponding call to `Put` followed by a call of the procedure `New_Line`.

The procedure `Get_Line` is designed primarily to read complete lines of input; it reads characters from the current 'read position' and stores them in the string supplied as the AP until it encounters an e-o-l marker whereupon the procedure 'returns' and leaves the read position just *after* the e-o-l marker (ie. at the start of the next input line). However, if the string is filled before the end of a line is reached, then `Get_Line` behaves just like `Get`. Note that, unlike the procedure `Get`, `Get_Line` is capable of reading a variable number of characters and in order that the program can determine how many characters have in fact been 'read', `Get_Line` also has a formal `OUT` parameter `Last` of type `Natural`, which is used to 'pass back' the number of characters read.

To illustrate the use of the `Get_Line` and `Put_Line` procedures, the following Ada fragment copies a complete file (assuming appropriate file connections) provided only that the file being copied has an e-o-l marker immediately before end-of-file :

```
MaxLineLength : CONSTANT Natural := 80;
SUBTYPE LineType IS String(1 .. MaxLineLength + 1);
Line : LineType;
LineLength : Natural;
....
WHILE NOT End_Of_File LOOP
   Get_Line(Item => Line, Last => LineLength);
   Put_Line(Item => Line(1 .. LineLength));
END LOOP;
```

Note the use of a slice in the `Put_Line` step to ensure that only the number of characters actually read by the `Get_Line` step are output. In the above fragment it is assumed that the maximum line length is 80 characters; `LineType` is declared to be capable of holding one extra character to ensure that the call of `Get_Line` always consumes the e-o-l marker (even on a line which contains 80 characters) and so moves to the start of the next line of the input file.

Important note

The procedures for string I/O described above require an AP of (a subtype) of the pre-defined type `String`. For example:

```
    SUBTYPE WordType IS String(1..20);
    MyWord : WordType;
    ..................
```

[3] There are no versions of the `Put_Line` procedure for the output of `Integer` or `Float` values.

```
Put(MyWord);
```

If we were to replace the subtype declaration with the declaration of a new character array type:

```
TYPE WordType IS ARRAY (1..20) OF Character;
```

the `Put` statement would give rise to a compilation error because Ada regards any user-defined character array type as distinct from the predefined type `String`.

15.8 Programming Example

It is required to process a user-specified file to remove surplus whitespace (blanks and blank lines) and store the processed output in a second user-specified file. Sequences of blanks separating words should be replaced by a single blank. Leading and trailing blanks on a line, empty lines and lines containing only blanks should all be removed.

Outline Algorithm

```
Prompt user for input and output files and open them ...... A
WHILE NOT End_Of_File LOOP
    Get ThisLine .......................................... B
    IF ThisLine not empty THEN ............................ C
        Remove surplus whitespace from ThisLine ........... D
        IF ThisLine is still not empty THEN ............... E
            Output ThisLine ............................... F
        END IF
    END IF
END LOOP
Close the input and output files .......................... G
```

We will need a string variable to hold the current line whilst we process it and also to communicate between stages **B**, **C**, **D**, **E** and **F**:

```
ThisLine  : String(1 .. MaxLineLength + 1);
```

Stages **A**, **B**, **C**, **E**, **F** and **G** are relatively straightforward. Step **D** is easily the most complicated and will be procedurised (for clarity) so step **D** will become the procedure call step:

```
Squeeze(Line => ThisLine(1 .. OrigLength), Length => NewLength);
```

where `OrigLength` is the length of the input line assigned in step **B**. The string `Line` will need to be an `IN OUT` parameter to pass in the original line and pass back the modified line. `Length` will need to be an `OUT` parameter to pass back the length of the modified line.

The programming of the body of `Squeeze` is facilitated by introducing a state variable of the enumeration type:

```
TYPE ProcessingState IS (InMargin, InWord, InSpacing);
```

As we process characters of the line, different actions are required depending on the current character and on the current state of processing; that is whether we are in leading spaces at the start of a line (`InMargin`), in the middle of a sequence of non-whitespace characters (`InWord`) or in 'inter-word' whitespace (`InSpacing`). For example, if we are in the state `InSpacing` and encounter a non-whitespace character the state should change to `InWord` and we should 'output' a single blank (for spacing) followed by the current character.

Full Program

```
WITH Ada.Text_IO; USE Ada.Text_IO;
WITH CS_File_IO;  USE CS_File_IO;

PROCEDURE Compress IS
   -- Program to remove surplus whitespace from a user-specified file.
   -- Written by Alan Barnes, November 12th, 1993 for ISP Ada course.
   -- Updated for Ada95 by Alan Barnes, August 1996.

   MaxLineLength : CONSTANT Positive := 132; -- Big enough for most files

   FUNCTION IsWhite (Ch : Character) RETURN Boolean IS
      Blank : CONSTANT Character := ' ';
   BEGIN
      RETURN Ch = Blank;
   END IsWhite;
   PROCEDURE Squeeze (Line : IN OUT String; Length : OUT Natural) IS
      TYPE ProcessingState IS (InMargin, InWord, InSpacing);
      State  : ProcessingState := InMargin;
   BEGIN
      Length := Line'First - 1;
      FOR InPos IN Line'Range LOOP
         CASE State IS
            WHEN InMargin =>
                IF NOT IsWhite(Line(InPos)) THEN
                   State := InWord;
                   Length := Length + 1;
                   Line(Length) := Line(InPos);
                END IF;
            WHEN InWord =>
                IF IsWhite(Line(InPos)) THEN
                   State := InSpacing;
                ELSE
                   Length := Length + 1;
                   Line(Length) := Line(InPos);
                END IF;
            WHEN InSpacing =>
                IF NOT IsWhite(Line(InPos)) THEN
                   State := InWord;
                   Line(Length + 1) := Blank; -- Single blank for spacing
                   Length := Length + 2;
                   Line(Length) := Line(InPos);
                END IF;
         END CASE;
      END LOOP;
   END Squeeze;

   ThisLine : String(1 .. MaxLineLength + 1); -- To hold current line
   OrigLength, NewLength : Natural; -- To hold its old & new lengths

BEGIN  -- Main Program of Compress
   Put_Line("Please enter the name of the file to compress.");
   OpenInput;
   Put_Line("Please enter a different name for the compressed file.");
   OpenOutput;

   WHILE NOT End_Of_File LOOP
      Get_Line(Item => ThisLine, Last => OrigLength);

      IF OrigLength > 0 THEN         -- Do nothing if line is empty
         Squeeze(Line => ThisLine(1 .. OrigLength), Length => NewLength);
         IF NewLength > 0 THEN  -- Do nothing if modified line is empty
            Put_Line(Item => ThisLine(1 .. NewLength));
         END IF;
      END IF;

   END LOOP;

   CloseInput;
   CloseOutput;
END Compress;
```

Note the use of string slices as APs in the calls of `Squeeze` and `Put_Line`, this causes the corresponding FP string parameters in these procedures to be 'full up'. Also note that the call to `Squeeze` is 'protected' in a selection so that it is not called with a empty slice (which would occur when there is an empty line in the input). Although this *may* not be strictly necessary (the body of the `FOR` loop in `Squeeze` would never be executed in this case), it seems safer and clearer to do so. This is an instance of 'defensive' programming.

Finally note that in procedure `Squeeze`, `InPos` is always greater than or equal to `Length`. Thus it is safe to overwrite the string as we go since the characters overwritten will already have been 'processed'. Actually the use of a string variable `ThisLine` is not strictly necessary here, and you might wish to consider re-writing this program so that it inputs characters one by one and outputs them directly to the output file as and when necessary, without the need for the variable `ThisLine`.