Introduction to Systematic Programming
Unit 13 - Discrete Types; Selection Using `CASE`

13.1 Enumeration types

Sometimes the pre-defined types available in Ada (eg. `Integer`, `Float`, `Boolean` and `Character`) are not convenient for expressing the actions we wish to perform. For example suppose we wanted to process some data involving the days in a week. We could refer to each day name by an integer code number, say, Monday = 1, Tuesday = 2, ..., Sunday = 7, and hence declare a variable to hold a particular day (code number) using:

```
Day : Integer;
```

and then include in our program steps of the form:

```
Day := 5;
```
or
```
IF Day = 3 THEN .....
```

But such steps are obscure (we have to refer back to our numbering system to recall that Friday = 5 and Wednesday = 3). It would be much more natural to be able to write `Friday` or `Wednesday` (or perhaps suitable abbreviations) directly into our program. Ada allows us to define a group of **new literal values**, ie. new data values (like the number value `6`, or the Boolean value `True`, or the character value `'A'`), by using an **enumeration type**. We can do this by writing in a type declaration the list of values (enclosed in round brackets) that variables of this type are capable of holding, eg:

```
TYPE DayOfWeek IS (Mon, Tues, Wed, Thurs, Fri, Sat, Sun);
```

To avoid confusion it is normal to choose identifiers for the enumerated literal values appearing in this definition to be different from the names of any variables, constants, procedures, types or other enumerated literals appearing elsewhere in the program.

In the same way that the declaration:

```
Number : Integer;
```

means that the variable `Number` can hold *one* of the possible integer values, ..., `-2`, `-1`, `0`, `1`, `2`, ..., the following declaration of the variable `Day` means that it can hold *one* of the seven listed values of the type `DayOfWeek`:

```
Day : DayOfWeek;
```

Following this form of declaration, we can perform assignment and comparison of such values, so that we can re-write the earlier program steps as:

```
Day := Fri;
```
or
```
IF Day = Wed THEN .....
```

Notice how such user-specified types improve the clarity of a program by removing the problem of artificially representing data values which are not amongst those of the pre-defined types.

As with declarations involving the pre-defined types, it is possible to define enumeration constants and have initialised enumeration variable declarations. For example:

```
TYPE Season IS (Winter, Spring, Summer, Autumn);
HolidaySeason : CONSTANT Season := Summer;
PresentSeason : Season := Autumn;
```

defines the enumeration constant `HolidaySeason` to be the value `Summer`, and an enumeration variable `PresentSeason`, capable of holding any one of the four enumeration values listed, but initialised to the value `Autumn`. It is also possible to have procedures and functions which have enumeration type parameters, and we will see examples of this later in the unit.

13.2 Using enumeration variables

The operations which can be performed on enumeration values are quite restricted. The standard arithmetic operators like +, -, *,... etc. are not defined for such values, eg:

```
        Day := Mon + Tues;    -- !? Illegal in Ada
```

does not have any sensible meaning if `Day` is to represent only one of the values `Mon`, `Tues`, `Wed`, ...,`Sun`. We may however compare two values of the *same* enumeration type by means of the standard relational operators =, /=, <, etc. The ordering is determined by the order in which the enumeration values are listed when the type is declared. Thus:

`Mon < Tues`	yields the value	`True`
`Fri <= Tues`	yields the value	`False`

Input and output of enumeration values is possible in Ada using versions of the procedures `Get` and `Put` which have been 'informed' about the type of value they are expected to input or output. We can create these versions of the `Get` and `Put` procedures by writing something like:

```
PACKAGE DayOfWeek_IO IS NEW Enumeration_IO (DayOfWeek);
USE DayOfWeek_IO;
```

which would be placed in the declaration part of an Ada program (after the declaration of the type `DayOfWeek`). The first of these steps creates a new package (called `DayOfWeek_IO`) containing versions of the `Get` and `Put` procedures specifically modified for the input and output of values of type `DayOfWeek`. The second step then makes the contents of this package available for use in the program without the need to precede each `Get` or `Put` with the package name. Note the absence of a `WITH` step in the above, since the first step not only creates a new package, but also makes it available to the program. Following this declaration we may then write input and output steps like:

```
Put("Type in a day of the week ");
Get(Day);
New_Line;
Put("You have just entered ");
Put(Day);
```

where the value input must be one of the listed values of the type `DayOfWeek` (expressed using a mixture of either upper or lower case characters), and will appear all in capital letter characters when output. Thus the above would produce the interactive conversation (where the user input is shown underlined):

```
              Type in a day of the week tues
              You have just entered TUES
```

Notice that if the enumeration values listed in the type declaration are abbreviated, it is the abbreviated values which have to be input to a `Get` step, and which are output by a `Put` step. If the 'capitalised' or abbreviated forms of output shown above are inconvenient, we can use another way of outputting enumeration values described later in this unit.

13.3 Discrete types

Collectively the pre-defined types `Integer`, `Character` and `Boolean`, and user-defined enumeration types are known as the **discrete types** (also sometimes referred to as **ordinal types**). They are referred to in this way because the values which can be represented in these types form an ordered discrete list of values. We have seen how this ordering is defined for enumeration values, but how is it defined for the type `Character`? Each value of type `Character` is represented internally in the computer by a numerical code. Usually the Latin-1 (ISO Standard 8859-1) is used where each character is represented by an 8-bit numerical code (ie. by an integer value in the range 0 to 255)[1] . Character values are ordered by the ordering of their Latin-1 codes. The full ordering can be obtained by inspecting a table of these codes and their corresponding characters - most programming textbooks contain such a table. However, it is unusual to use this ordering for anything other than the upper and lower case letter characters and the digit characters, where the ordering is:

a)	Upper case letters	`'A'` < `'B'` < `'C'` < `'D'` < ... < `'Z'`
b)	Lower case letters	`'a'` < `'b'` < `'c'` < `'d'` < ... < `'z'`
c)	Digit characters	`'0'` < `'1'` < `'2'` < `'3'` < ... < `'9'`

[1] Latin-1 character codes in the range 0 to 127 coincide with the codes in the well-known ASCII system (American Standard Code for Information Interchange). However the latter should be regarded as obsolescent.

Thus when character values are compared using the relational operators <, >, >= etc; the ordering of characters is determined by the numerical order of the codes used to represent them.

Discrete types are quite important in programming in Ada because:

i) Consecutive values from the ordered list of values comprising each of these types can be used as array subscripts.

ii) Consecutive values from the ordered list of values comprising each of these types can be used to 'count' in a `FOR` repetition.

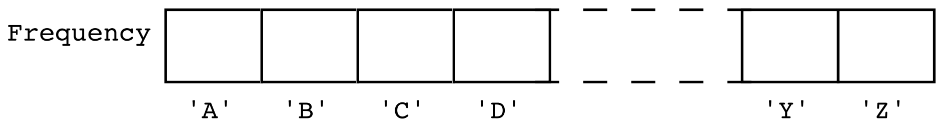
We have seen examples of these features in earlier units for `Integer` values, so we now give examples for the `Character` and enumeration discrete types.

`Character` types

i) To declare a 26 element array, subscripted using character values in the range `'A'` to `'Z'`, we would write:

```
TYPE LetterArray IS ARRAY ('A'..'Z') OF Integer;
Frequency : LetterArray;
```

which declares an array variable `Frequency`, whose elements can hold `Integer` values, and which are indexed (or subscripted) using values in the range `'A'` to `'Z'`, eg. an array of the form:



For example to read a sentence of text (comprising blanks and capital letter characters) terminated by a full-stop character, and to count the frequency of each capital letter, using the elements of this array to hold these counts, we would write:

```
Get(NextChar);
WHILE NextChar /= '.' LOOP
   IF NextChar /= ' ' THEN
      Frequency(NextChar) := Frequency(NextChar) + 1;
   END IF;
   Get(NextChar);
END LOOP;
```

which uses a `Character` variable `NextChar`.

ii) Then to output a table of these frequencies, we would write:

```
Put("Letter  Frequency");
FOR ThisChar IN 'A'..'Z' LOOP
   New_Line;
   Put(ThisChar);
   Put(Item => Frequency(ThisChar), Width => 12);
END LOOP;
```

where we see that on each repetition of the `FOR` loop, the index variable `ThisChar` takes in turn the values `'A'`, `'B'`, `'C'`, ..., `'Z'`. Thus the first time around the repetition the `Put` steps output the character `'A'` and the array element `Frequency('A')`, the second time around the repetition they output `'B'` and `Frequency('B')`, and so on.

Enumeration types

i) To declare a five element array, subscripted using enumeration values in the range `Mon` to `Fri`, we would write:

```
TYPE WorkingHours IS ARRAY (Mon..Fri) OF Float;
HoursWorked : WorkingHours;
```

which declares an array variable `HoursWorked`, whose elements can hold `Float` values, and which are indexed (or subscripted) using the values `Mon` to `Fri` from the enumeration type `DayOfWeek` defined earlier, eg. an array of the form:

HoursWorked	8.5	7.5	8.0	8.25	7.5
	Mon	Tues	Wed	Thurs	Fri

ii) To prompt the user to input five values for the elements of this array we would write:

```
FOR ThisDay IN Mon..Fri LOOP
   Put("Input the number of hours worked on ");
   Put(ThisDay);
   Get(HoursWorked(ThisDay));
   New_Line;
END LOOP;
```

which would produce an interactive conversation something like:

```
Input the number of hours worked on MON   8.5
Input the number of hours worked on TUES  7.5
Input the number of hours worked on WED   8.0
Input the number of hours worked on THURS 8.25
Input the number of hours worked on FRI   7.5
```

and would result in storing the values shown in the array diagram above.

13.4 Operations on discrete types

Since the values of all the discrete types are ordered, the notion of an immediate successor and predecessor of a value clearly make sense. For example, for the enumeration type `DayOfWeek` defined above, the predecessor and successor of the value `Tues` are `Mon` and `Wed` respectively. As another example, the character `'b'` has `'a'` as its predecessor and `'c'` as its successor. We see that in general each value (except the first) has an immediate predecessor and each value (except the last) has an immediate successor, determined by the order in which the values are listed when the enumeration type was defined.

A number of useful built-in operations are provided for manipulating and converting values belonging to these discrete types. Many of these operations (including the ones below) are defined as function **attributes** of a type. So that they are invoked by stating both the type name, and the particular attribute required using:

`Type_name ' Function_name ( Expression )`

or

`Type_name ' Function_name`

Predecessor and successor functions

The function `Succ` delivers the successor of the value specified in the expression. For example, for the enumeration type `DayOfWeek` defined above, assuming the declaration:

```
Day : DayOfWeek := Thurs;
```

the step:

```
Day := DayOfWeek'Succ(Day);
```

will set the value of `Day` to `Fri`. If the value of `Day` is the maximum for that particular type, (ie. if `Day` holds the value `Sun`) then executing `Succ(Day)`will result in a run-time error.

The companion function `Pred` delivers the predecessor of the value specified in the expression. For example, if `Day` held the value `Thurs` then the step:

```
Day := DayOfWeek'Pred(Day);
```

will set the value of `Day` to `Wed`. If the value of `Day` is the minimum for that particular type, (ie. if `Day` holds the value `Mon`) then executing `Pred(Day)`will result in a run-time error.

Position and value functions

The function `Pos` delivers the position of the value as specified in the ordered discrete list of values. For example:

`Character'Pos('A')`	yields the value `65` (the Latin-1 code for `'A'`)
`DayOfWeek'Pos(Tues)`	yields the value `1` (enumeration values are numbered from 0)

The companion function `Val` delivers the value of the discrete type which appears in the specified position. For example:

`Character'Val(49)`	yields the value `'1'` (as the Latin-1 code for `'1'` is `49`)
`Character'Val(32)`	yields the value `'Δ'` (as the Latin-1 code of blank is `32`)
`DayOfWeek'Val(3)`	yields the value `Thurs` (enumeration value numbered `3`)

To illustrate the use of the `Pos` and `Val` functions, consider the following function to convert letter characters from lower to upper case[2]:

```
FUNCTION UpperCase (Ch : Character) RETURN Character IS
   -- To convert lower case letters to upper case. All
   -- non-letter characters are left unaltered
   Shift : CONSTANT Integer := Character'Pos('A')
                                        - Character'Pos('a');
BEGIN
   IF Ch >= 'a' AND Ch <= 'z' THEN
       RETURN Character'Val(Shift + Character'Pos(Ch));
   ELSE
       RETURN Ch;
   END IF;
END UpperCase;
```

First and last functions

The parameterless functions `First` and `Last` deliver the first and last values respectively in the ordered discrete list of values for that type. For example:

`DayOfWeek'First`	yields the value `Mon`
`DayOfWeek'Last`	yields the value `Sun`
`Character'First`	yields the "null" character (with Latin-1 code of 0)
`Character'Last`	yields the character ÿ (with Latin-1 code of 255)

Notice that although all of these functions apply to the discrete type `Integer`, some are not very useful. For example, assuming the `Integer` variable `Count`, then the steps:

```
Count := Integer'Succ(Count);       Count := Count + 1;
```

are equivalent, though we wouldn't choose to use the first on the grounds of program clarity!

13.5 Multiple selection using CASE

We have seen that Ada provides one-way selections by the use of `IF...THEN...END IF`, two-way selections by the use of `IF...THEN...ELSE...END IF` and more complicated selections by the use of an `IF...THEN...ELSIF....ELSE...END IF` step, or by using several consecutive `IF`'s. For example, suppose we wanted to output a description of whether or not shops are open on a particular day using one of the above we could write this as:

```
IF Day = Sun THEN
     Put("Shops are closed all day");
ELSIF Day = Fri OR Day = Sat THEN
     Put("Shops are open all day");
ELSIF Day = Thurs THEN
    -- Early closing day is on Thursday
     Put("Shops are open half a day");
ELSE
    Put("Shops are open all day");
END IF;
```

2 This function does not convert accented lower-case characters (such as é) to their upper-case equivalents.

Generally, the greater the number of `ELSIF` parts there are, the more obscure the selection process becomes, and hence the more error-prone.

Ada provides a much more convenient way of expressing multiple selections known as a `CASE` step. The `CASE` form equivalent to the above example would be:

```
CASE Day IS
   WHEN Sun              => Put("Shops are closed all day");
   WHEN Mon..Wed|Fri|Sat => Put("Shops are open all day");
   WHEN Thurs            => Put("Shops open for half a day");
END CASE;
```

13.6 General form of a `CASE` step

The general structure of a `CASE` step is typified by:

```
CASE Selecting_expression IS
     WHEN Case_list$_1$ => Step(s)$_1$
     WHEN Case_list$_2$ => Step(s)$_2$
     ........................
     WHEN Case_list$_n$ => Step(s)$_n$
END CASE;
```

where each *`Case_list`* stands for a list of values separated by vertical bar characters. A *`Case_list`* may also include a range of consecutive values using the conventional notation, namely *`First`*`..`*`Last`* which denotes all the range of values from *`First`* to *`Last`* inclusive. One of the *`Case_List`*'s may consist of the keyword `OTHERS`.

The *`Selecting_expression`* must be a variable (or expression) of one of the discrete types, and each *`Case_list`* is a list of some of the values that the *`Selecting_expression`* can yield when evaluated. The meaning of such a `CASE` step is:

i) Evaluate the *`Selecting_expression`*
ii) If the value of the *`Selecting_expression`* matches one of the values in *Case_list*$_i$, perform the corresponding *Step(s)*$_i$
iii) Otherwise perform the steps in the `OTHERS` part.

The `OTHERS` part is optional but if it is omitted all possible values of the *`Selecting_expression`* *must* occur in one or other of the *`Case_list`*'s, otherwise a compilation error will occur. The Ada compiler also checks that each possible value of the *`Selecting_expression`* occurs *once and once only* amongst all values given in the *`Case_list`*'s.

The example shown earlier uses an enumeration type value. As another example we could use `Character` values to determine the score appropriate to any letter in the game of Scrabble™ by:

```
CASE Letter IS
   WHEN 'D'|'G'             => Score := 2;
   WHEN 'B'|'C'|'M'|'P'     => Score := 3;
   WHEN 'F'|'H'|'V'|'W'|'Y' => Score := 4;
   WHEN 'K'                 => Score := 5;
   WHEN 'J'|'X'             => Score := 8;
   WHEN 'Q'|'Z'             => Score := 10;
   WHEN OTHERS              => Score := 1;
END CASE;
```

assuming declarations of the variables `Score` and `Letter` of type `Integer` and `Character` respectively, and `Letter` holds one of the values `'A'` to `'Z'`. We see that in such multiple selections the `CASE` step provides a much neater way of expressing the actions to be performed. Imagine writing this using the logical operators `AND` and `OR` in compound conditions in an `IF...THEN...ELSIF...ELSE...END IF` step!

We can also use a `CASE` step to produce a more convenient form of output of the values of an enumeration type. If we don't want to have the capitalised form shown at the end of [13.2], or if we don't want to output the abbreviated form of the enumeration literals, we can use instead:

```
PROCEDURE PutDayOfWeek (ThisDay : IN DayOfWeek) IS
   -- To output a suitable text value for the day ThisDay.
BEGIN
   CASE ThisDay IS
      WHEN Mon   => Put("Monday");
      WHEN Tues  => Put("Tuesday");
      WHEN Wed   => Put("Wednesday");
      WHEN Thurs => Put("Thursday");
      WHEN Fri   => Put("Friday");
      WHEN Sat   => Put("Saturday");
      WHEN Sun   => Put("Sunday");
   END CASE;
END PutDayOfWeek;
```

13.7 Programming example - age distribution survey

The ages (in years) of a sample of the children in the population have been prepared in a file as input data, terminated by the value -1. It is required to determine and output the number of children in each of the age categories.

a) Pre-school age (0-2 years)
b) Play group age (3-4 years)
c) Primary school age (5-11 years)
d) Secondary school age (12-16 years)
e) Sixth form age (17-18 years)

First thoughts:

Read in the data values one-by-one, adding 1 to the appropriate count (one for each age category). Loop terminated by reading the value -1.

Data structures:

We could use five separate variables for the counts, or an array:

```
TYPE GroupTotals IS ARRAY (1..5) OF Integer;
Total : GroupTotals;
```

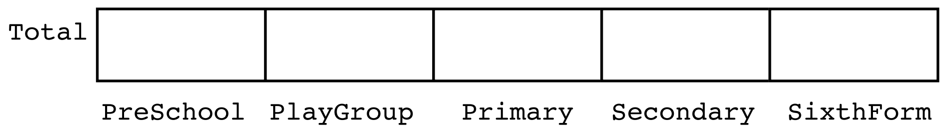
but it would not be clear in our program which element of the `Total` array was used as the count for a particular age category. A better approach would be to define an enumeration type, capable of holding a value denoting each of the five age categories:

```
TYPE AgeGroup IS (PreSchool,PlayGroup,Primary,Secondary,SixthForm);
```

and an array of counts, subscripted by these enumerated values defined as follows:

```
TYPE GroupTotals IS ARRAY (PreSchool..SixthForm) OF Integer;
Total : GroupTotals;
```

ie. an array of the form:



where elements of the array are used to hold counts of the number of children in each age category.

Example output:

A typical layout of output values should be something like:

```
  Age Range       No. of Children
 0 -  2 years           15
 3 -  4 years           33
 5 - 11 years          104
12 - 16 years          146
17 - 18 years           13
```

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

PROCEDURE AgeSurvey IS
    -- Program for Unit 13 of ISP Ada course.
    -- Written by L J Hazlewood, October 1993.
    -- Modified for Ada 95 by A Barnes, August 1996.

    TYPE AgeGroup IS (PreSchool,PlayGroup,Primary,Secondary,SixthForm);
    TYPE GroupTotals IS ARRAY (PreSchool..SixthForm) OF Integer;
    Terminator : CONSTANT Integer := -1;

    FUNCTION AgeRange (ThisAge : Integer) RETURN AgeGroup IS
        -- To determine the age category for ThisAge.
    BEGIN
        CASE ThisAge IS
            WHEN 0..2   => RETURN PreSchool;
            WHEN 3|4    => RETURN PlayGroup;
            WHEN 5..11  => RETURN Primary;
            WHEN 12..16 => RETURN Secondary;
            WHEN 17|18  => RETURN SixthForm;
            WHEN OTHERS => Put("Error in data, invalid age"); New_Line;
               -- This last Put step should never be executed. If it is,
               -- a run-time error occurs as AgeRange returns no value.
        END CASE;
    END AgeRange;

    PROCEDURE OutputAgeRange (ThisGroup : IN AgeGroup) IS
        -- To output the range of ages for ThisGroup age category.
    BEGIN
        CASE ThisGroup IS
            WHEN PreSchool => Put(" 0 -  2 years");
            WHEN PlayGroup => Put(" 3 -  4 years");
            WHEN Primary   => Put(" 5 - 11 years");
            WHEN Secondary => Put("12 - 16 years");
            WHEN SixthForm => Put("17 - 18 years");
        END CASE;
    END OutputAgeRange;

    PROCEDURE OutputTable (Count : IN GroupTotals) IS
        -- Output a table showing the age ranges for each age category
        -- and the corresponding number of children in the sample.
    BEGIN
        New_Line(4);
        Put("  Age Range        No. of Children");  New_Line;
        FOR Group IN PreSchool..SixthForm LOOP
            OutputAgeRange(Group);
            Put(Item => Count(Group), Width => 16);  New_Line;
        END LOOP;
    END OutputTable;

    Total      : GroupTotals := (PreSchool..SixthForm => 0);
    Age        : Integer;
    Category   : AgeGroup;

BEGIN  -- Main program.
    Put("Please type in the name of the data file"); New_Line;
    OpenInput;
    Get(Age);
    WHILE Age /= Terminator LOOP
        Category := AgeRange(Age);
        Total(Category) := Total(Category) + 1;
        Get(Age);
    END LOOP;
CloseInput;
    OutputTable(Total);
END AgeSurvey;
```