Introduction to Systematic Programming
Unit 12 - The Type `Character`. More about Input and Output

12.1 Introduction

Whether input to a program is typed directly at a VDU or first typed into a file and then "read" by the program from the file, it consists quite simply of a sequence of characters. Characters fall into a number of groups, for example:

a)	alphabetic	- upper case	`A B C ... Z`
		- lower case	`a b c ... z`
b)	digits		`0 1 2 ... 9`
c)	punctuation		`. , ; : ? !`
d)	symbols		`+ - * / < > =`
e)	space or blank - conventionally shown as		Δ
f)	special "non-printing" characters		`Return`, `Tab` and other "control" characters
g)	characters for various European languages etc.		`é ä Å ¿ ¡`

The input that we have looked at so far has contained only digits, the blank and `Return` characters (to separate numbers), '`.`' as a decimal point and '–' to indicate negative numbers. Furthermore, unbroken sequences of digit characters were treated in a special way - in fact they are regarded as representations of denary (ie. base 10) numbers - so that when (groups of) digit characters are read in they are **converted** to (the internal representation, usually binary, used by the computer when storing) integers or real numbers, for example:

```
IntVar : Integer;
...
Get(IntVar);  -- Using Get from CS_Int_IO
...
```

input data		**inside computer**		
`17`	(2 characters)	`IntVar`	`00...0010001`	(17_{10} in binary)

The characters in `17` above are converted into an internal `Integer` number because the procedure `Get` (from the library package `CS_Int_IO`) expects to find characters representing an `Integer` number in the input.

However we may not always want characters converted on input; instead we may want the individual characters "as they are", or the data may be such that no numeric conversion makes sense, for example textual input. If we assume the above declarations and attempt to execute the program step: `Get(IntVar)` with the data value `X`, we will get a run-time error since the character `X` is not a valid `Integer`. To input individual characters as such, without any conversion, we need to introduce a new Ada data type, namely the type `Character`.

12.2 Using character variables

Character variables can be declared along with other variables in the usual way, for example:

```
IntVar              : Integer;
First, Second, Sign : Character;
```

define a variable `IntVar` capable of holding an `Integer` value, and three variables `First`, `Second` and `Sign` each capable of holding a single character. Character values may be input using a procedure `Get` imported from the library package `Ada.Text_IO`. Hence any program which uses the procedure `Get` for inputting character values should include the context clauses:

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

Assuming that the input data `17` is typed at the keyboard, only a single "`Get` integer" call is required to consume the data whereas two calls of "`Get` character" are required to consume the same data, ie:

input data			inside computer	
	`Get(IntVar);`	produces	`IntVar`	`00...0010001`
`17`	whereas			
	`Get(First);`	produces	`First`	`'1'`
	`Get(Second);`		`Second`	`'7'`

The **character values** `'1'` and `'7'` are shown in quotes to indicate particular literal characters rather than any sort of number. Blanks are skipped when reading into numeric variables, but are "full-status" characters when input to a `Character` variable is performed, for example:

input data			inside computer	
	`Get(IntVar);`	produces	`IntVar`	`00...0010001`
`Δ17`	whereas			
	`Get(Sign);`	produces	`Sign`	`'Δ'`
	`Get(First);`		`First`	`'1'`
	`Get(Second);`		`Second`	`'7'`

We can also give character values to character variables with assignment steps, for example:

```
First := 'X';   Sign := '+';   Second := 'Y';
```

Note that the (single) quotes are mandatory and that they must enclose precisely one character - a simple character variable can store just one character. If the quote marks were omitted, the assignment:

```
First := X;
```

would mean copy the *contents* of the *character variable* `X` (with the computer assuming that `X` has been declared to be of type `Character`) into the character variable `First`.

The values of character variables can also be output using a procedure `Put` which is imported from the package `Ada.Text_IO`, for example, after the above assignment steps then:

```
Put(First);
Put(Sign);      produces the output:      X+Y
Put(Second);
```

12.3 Example

Given a file containing data consisting of a single name in the *precise* form:

forenameΔforenameΔsurname.

(ie. exactly two forenames followed by one surname) terminated by a full stop, output it to another file after converting it to a new form as illustrated by the following:

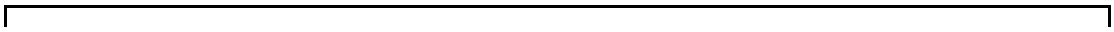
input data	required output
`MARGARETΔHILDAΔTHATCHER.`	`THATCHER,ΔM.H.`

Assuming the declarations:

```
Ch       : Character;  -- Next input character
Initial1 : Character;  -- Initial of first forename
Initial2 : Character;  -- Initial of second forename
```

a suitable Ada fragment would be:

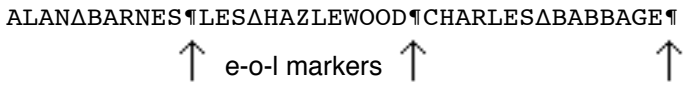
```
OpenInput;
OpenOutput;

Get(Initial1);           -- Input first initial.

Get(Ch);
WHILE Ch /= ' ' LOOP
   Get(Ch);              -- Skip rest of first forename
END LOOP;                -- until next blank is input.
```



```
Get(Initial2);           -- Input second initial.

Get(Ch);
WHILE Ch /= ' ' LOOP
   Get(Ch);              -- Skip rest of second forename
END LOOP;                -- until next blank is input.

Get(Ch);                 -- Input first character of surname.

WHILE Ch /= '.' LOOP     -- Copy surname characters to the output
   Put(Ch);              -- until a full-stop is input.
   Get(Ch);
END LOOP;

Put(", ");               -- Output a comma and space.
Put(Initial1);
Put('.');                -- Output the two initials.
Put(Initial2);
Put('.');
New_Line;

CloseInput;
CloseOutput;
```

12.4 The structure of files

The example of [12.3] was easily solved because a full-stop was used to mark the end of a complete name, but what could we do if there were no such marker? For example, consider a file of names where each complete name is on a separate line, but where there is no full-stop at the end of each surname. To answer this we must see how files are structured. Ada treats a file as a sequence of lines, each made up of individual characters, and where each line is *terminated* by a special **end-of-line (e-o-l) marker**; for example the following could be part of a file of names, arranged one full name per line (for simplicity we consider names consisting of a single forename and surname only):



When outputting to a disc file a call to the procedure `New_Line` will cause an e-o-l marker (shown as ¶) to be "written" to the file concerned. Outputting an e-o-l marker to a VDU causes the cursor position to advance to the start of a new line on the VDU screen. When inputting from a keyboard, we can consider that an e-o-l marker is entered each time the `Return` (or "new-line") key is pressed.

12.5 Detecting the end of a line of input

Ada allows us to determine when an e-o-l marker has been reached, ie. to be able to detect the end of a line of input, by using the function `End_Of_Line` imported from the `Ada.Text_IO` library package. When this Boolean-valued function `End_Of_Line` is called, it returns `True` if we are at the end of a line (ie. if the current "read position" is immediately before the e-o-l marker) and `False` otherwise. `End_Of_Line` works equally well whether input is being taken from the keyboard or from a file (following an `OpenInput` command). Note that `End_Of_Line` is a *function* which takes no parameters. Note also that we do not need to know what the actual e-o-l marker is in order to use `End_Of_Line` (in any case this marker may vary from one computer system to another).

Thus, for example, to count the number of characters in a particular line we would write the Ada fragment:

```
NextChar   : Character; -- The next character input.
NumChars   : Integer;   -- To count the number of characters.
......
NumChars := 0;
WHILE NOT End_Of_Line LOOP   -- Continue until e-o-l marker.
   Get(NextChar);            -- Input the next character.
   NumChars := NumChars + 1; -- Increment the count.
END LOOP;
```

Note that when using this form of data processing loop it is not necessary to input the first data value from outside the loop; the Boolean value returned by `End_Of_Line` (rather than some condition based on already input data values) is used to control the loop.

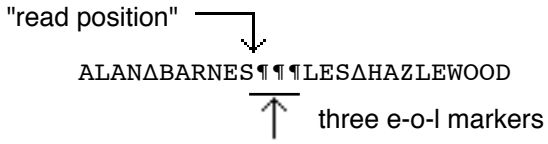
Also the surname processing part of [12.3] *without* the need for the `'.'` marker could now be coded thus:

```
WHILE NOT End_Of_Line LOOP
   Get(NextChar);
   Put(NextChar);
END LOOP;
```

where the (implicit) e-o-l marker effectively replaces the (explicit) full-stop character.

12.6 How do the various `Get` procedures deal with the e-o-l marker?

If the "read position" is at the end of a line (ie. immediately before the e-o-l marker) the `Get` procedure for performing character input skips over any end-of-line markers until a character is found. Thus if the input stream and current "read position" are as shown below, then a call of `Get(Ch)` causes the three end-of-line markers to be skipped and the character `'L'` to be stored in the variable `Ch` (which is assumed to be of type `Character`):



The `Get` procedures for performing numeric input all behave in the following manner:

i) They skip over any leading **white-space** (ie. blanks, tab characters and e-o-l markers) up to the first character of the number.

ii) Next they input the characters of the number (digits and certain other characters allowed in numbers such as `'+'`, `'-'` or `'.'`), and convert them into an internal computer representation for the number. This continues until a white-space character (or some other character which is not allowed in numbers) is reached; such a character thus terminates the number. This means in particular that numbers cannot extend from one line to the next.

If the characters input do not form a valid number (of the type `Integer` or `Float` expected) then a run-time error occurs.

Following a numeric input step, the "read position" will end up immediately *before* the terminating character. In particular if the number input was the last value on a line, the "read position" will be immediately before the e-o-l marker of that line.

The above now fully explains the behaviour of successive `Get` steps for numeric data values. Effectively it does not matter how such data values are laid out. Data could have been typed one value per line, or several values to a line separated by one or more spaces or tab characters, as each successive `Get` step will skip over any leading white-space characters until it finds the start of the next number. This is still the case if blank lines are inserted to make the data double-spaced.

If, for example, we are given the input data:

```
4621∆∆∆¶¶∆∆5678¶¶∆∆∆∆∆2349¶
```

we may simply input these values (assuming `First`, `Second` and `Third` are `Integer` variables) using the steps:

```
Get(First);
Get(Second);
Get(Third);
```

After the first `Get` step the "read position" is just after the character `1`, the second `Get` step will skip over the three remaining blanks on the line, the two e-o-l markers and the two leading blanks and the value `5678` will be input and stored in the variable `Second`. The "read position" will now be situated just after the digit `8`. The third `Get` step will skip over the two e-o-l markers and the 5 blanks and will

input the value 2349 and store it in the variable Third, leaving the "read position" just before the final e-o-l marker.

Although the most frequent terminating characters for numerical input are white-space (blanks, tabs and e-o-l markers), other characters can end a numeric data value. For example, given the input 533H (PAYE income tax codes are in this form), the input steps:

```
Get(TaxCode);
Get(TaxLetter);
```

would input the integer value 533 and store it in the variable TaxCode (assumed to be of type Integer). The character 'H' would act as the terminating character and would be available to be "read" by the second Get step which would store this character value in the variable TaxLetter (assumed to be of type Character).

12.7 Moving to a fresh line of input

Occasionally we may find it necessary to "skip over" any remaining data values on a line. For example, this might occur in an interactive program when we discover an error in one of the data values on a line, and we wish to report this error and request the program user to type the offending line again. In such a case we cannot leave the "read position" at its current location, since the next read step would simply continue to read the remainder of the offending line. Rather we must advance the "read position" to the end of the line. This could be achieved by skipping over individual characters using:

```
WHILE NOT End_Of_Line LOOP
   Get(Ch);
END LOOP;
```

This would leave the "read position" immediately *before* the e-o-l marker so that any subsequent Get step would skip the e-o-l marker and take its input from the start of the next line.

However the library package Ada.Text_IO contains a procedure Skip_Line which may be used to achieve virtually the same effect, namely: advance the "read position" from *wherever* it is in the current line to immediately *after* the next e-o-l marker, ie. to the beginning of the next line. A subsequent Get step would then take its input from this position.

Thus the Ada fragment above could in effect be replaced by a single procedure call step:

```
Skip_Line;
```

Actually the procedure Skip_Line has a formal IN parameter Spacing of type Integer which has the default value 1. The effect of a procedure call step such as:

```
Skip_Line(Spacing => 3);
```

is to advance the "read position" from *wherever* it is in the current line to immediately *after* the third e-o-l marker. Of course, omitting this parameter means that the default value of 1 is used and the "read position" is advanced to the beginning of the next line. In practice values of Spacing other than 1 are rarely used when Skip_Line is called and so this parameter is usually omitted.

12.8 Detecting the end of an input file

If we repeatedly "read" data values from a file, sooner or later we will reach the end of the file and if we attempt further "reads", a run-time error will be generated (as we are trying to read non-existent data). Ada provides a way of detecting when we have reached the end of the file so that we may avoid "running-off" the end of the file. We can determine when **end-of-file** has been reached by using the function End_Of_File imported from the library package Ada.Text_IO. When this Boolean-valued function End_Of_File is called, it returns True if the "read position" is at the end of the file[1] and False otherwise. Like the function End_Of_Line, End_Of_File takes no parameters.

In all the data processing examples that we have seen so far in these units, the number of data values has been known in advance (or has been given as the first item of data) or there has been a explicit data terminator (usually an "artificial" data value) at the end of the data. In the first of these cases the data should be processed using a FOR loop and in the second with a WHILE loop of the form:

---

[1] If a file ends with an e-o-l marker, End_Of_File also returns True when the "read position" is immediately before the last e-o-l marker in the file.

```
Get(DataValue);
WHILE DataValue /= Terminator LOOP
   Process this DataValue
   Get(DataValue);
END LOOP;
```

However, because we have a way of detecting end-of-file in Ada, it is not necessary to structure data files in this way. A file may contain an unspecified number of data values, but no explicit terminator need be included at the end of the data because we can call the function `End_Of_File` and use the value it returns to control our data processing loop. Using this function we would write our modified data processing loop as follows:

```
WHILE NOT End_Of_File LOOP
   Get(DataValue);
   Process this DataValue
END LOOP;
```

Here too, it is not necessary to input the first data value from outside the loop.

Note that we not need to know the precise mechanism used to detect whether end-of-file has been reached (in fact this may vary from one computer system to another[2]), we simply call the `End_Of_File` function to determine whether we have reached the end of the file. This enables Ada programs to run on many different types of computer system without modification.

Note however that in general it is more robust to test for end-of-line and skip to the start of the next line before testing for end-of-file as follows:

```
WHILE NOT End_Of_File LOOP
   WHILE NOT End_Of_Line LOOP
      Get(DataValue);
      Process this DataValue
   END LOOP;
   Skip_Line;
   Perform any end-of-line processing
END LOOP;
```

To see why this is necessary, consider the first of the latter two fragments of Ada given above. In the case when it is processing a file which ends with one or more empty lines (ie. there are consecutive e-o-l markers just before the end-of-file). When the "read position" is at the end of the last non-blank line in the file the test `End_Of_File` returns `False` as there is more than one e-o-l marker between the "read position" and the end of the file. Then the next `Get` command skips over all the e-o-l markers and attempts to read past end-of-file and a run-time error occurs. To avoid this we must use a nested loop to detect end-of-line and use `Skip_Line` to advance to the start of the next line before testing for end-of-file.

In Ada there should always be an e-o-l marker immediately before end-of-file (so that we cannot reach end-of-file "in the middle" of this line). To try and enforce this restriction, when an output file is closed using the procedure `CloseOutput`, an e-o-l marker will always be automatically inserted at end-of-file if one is not already present[3] . In these units we will assume that there will always be an e-o-l marker present immediately before the end-of-file. Given this and assuming that `DataValue` is of type `Character` the above nested loop will successfully process the file one character at a time even if it contains empty lines.

If `DataValue` is of type `Integer` or `Float` additional restrictions on the data file are required to ensure that the nested loop above processes the file correctly: namely there must be no trailing

---

[2] On some computer systems a special end-of-file marker is inserted at the physical end of a file. In Unix however, there is no physical end-of-file marker, instead end-of-file is detected by comparing the current "read position" measured in bytes from the start of the file with the known size of the file in bytes.

[3] Some utilities (editors etc.) which create files also obey this convention. However, some programs allow files to be created where there is no e-o-l marker immediately before the end-of-file.

white-space on lines in the input file (ie. no blanks and/or tabs immediately before the e-o-l marker) and similarly there must be no blank lines containing only white-space. To see why this is necessary consider the case when the "read position" is just before some trailing white-space on the last line of a file. The test `End_Of_Line` returns `False` as there is white-space between the "read position" and the e-o-l marker, then the next numeric `Get` command skips over the white-space and the e-o-l marker and attempts to read past the end-of-file and a run-time error occurs. Trailing white-space or blank lines elsewhere in the file do not cause run-time errors; however they do cause the inner loop to continue (as the test `End_Of_Line` returns `False`) and so the next numeric `Get` step consumes the white-space and the e-o-l marker until it finds valid numeric data and so the step(s):

```
Perform any end-of-line processing
```

would not be performed at the end of each line of the input file.

For the above reasons it is generally best to structure data files so that:

a) there is always an e-o-l marker immediately before the end of a file,

b) there are no empty or all blank lines (particularly at the end of the file), and

c) there is no trailing white-space on a line (particularly on the last line of the file).

As an example of the use of the functions `End_Of_File` and `End_Of_Line` consider the following Ada fragment which copies lines of one file to another preserving any blank lines:

```
Ch : Character;
..........
OpenInput("precious.dat");
OpenOutput("precious.bak");

WHILE NOT End_Of_File LOOP
   WHILE NOT End_Of_Line LOOP
      Get(Ch);   -- Read next char
      Put(Ch);   -- Copy it to back-up file
   END LOOP;
   Skip_Line;    -- Move to start of next line in input file
   New_Line;     -- Write e-o-l marker to output (back-up) file
END LOOP;
CloseInput; CloseOutput;
```

Note that it is necessary to use a nested `WHILE` loop which terminates at the end of each line in the input file followed by calls to `Skip_Line` and `New_Line` in order to ensure that the appropriate number of e-o-l markers are copied to the output file. If we were to write the main copying loop as:

```
WHILE NOT End_Of_File LOOP
    Get(Ch);   -- Read next char
    Put(Ch);   -- Copy it to back-up file
END LOOP;
```

then all the regular characters of the file would be copied correctly, but the output file would consist of a single line since no e-o-l markers would have been "written" to it.

You should also consider what would happen if the `Skip_Line` call step were omitted from the first program fragment above.

The function `End_Of_File` may also be used when inputting from a VDU/keyboard. At first sight this may seem surprising; however it is often useful to be able to signal to a program that the end has been reached of a sequence of data values which are being input interactively from the keyboard. The way of "typing end-of-file" from the keyboard depends on the particular computer system being used; in Unix this is achieved by typing `Control-D` (ie. pressing the `Control` and `D` keys together).

12.9 Arrays of characters

Arrays of characters can be declared in the usual way, for example:

```
MaxLineLength : CONSTANT Integer := 120;
TYPE LineType IS ARRAY (1 .. MaxLineLength) OF Character;
Line : LineType;
```

and, as with other arrays, each element of a `Character` array behaves like a single variable holding a single `Character` value, which can be assigned to, tested for a particular value, etc. We shall also see that character arrays provide a means for storing strings (or textual values) within a program.

For example, consider the following Ada fragment which gets a complete line of input from the keyboard, modifies a few characters and then outputs the modified line back to the VDU:

```
CharCount : Integer;
...................
CharCount := 0;
WHILE NOT End_Of_Line LOOP
   CharCount := CharCount + 1;
   Get(Line(CharCount));   -- Get next char and store it in array.
END LOOP;
New_Line;
Line(5)  := 's';
Line(7)  := 'B';
Line(11) := 's';
-- Now output modified line.
FOR I IN 1 .. CharCount LOOP
   Put(Line(I));
END LOOP;
New_Line;
```

It is left as an exercise for you to work out the output produced when the program fragment is given the input:

HelloΔHello!¶

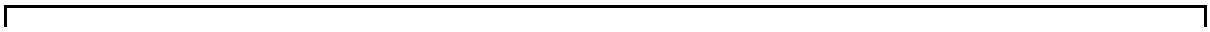
12.10 Programming example

A file consists of an unknown number of lines each of which consists of a surname followed by a series of forename initials, with an arbitrary number of blanks before and after the surname and between each initial. It is required to rearrange the name so that the initials appear before the surname, each initial followed by a full stop and a single blank. For simplicity we will assume that there are no blank lines in the file and that there are no trailing blanks at the end of any line.

line of input data	**output required**
ΔΔSmithΔΔΔMΔJΔΔΔΔΔΔK¶	M.ΔJ.ΔK.ΔSmith

Final Program

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

PROCEDURE NameChange IS
   -- Example program for Unit 12 of the ISP Ada course to
   -- illustrate character-based I/O.
   -- Written by A. Barnes,  October 1993
   -- Updated by L J Hazlewood,  October 1994
   -- Updated for Ada 95 by A Barnes,  August 1996

   MaxWordLength  : CONSTANT Integer   := 30;
   Blank          : CONSTANT Character := ' ';
   FullStop       : CONSTANT Character := '.';

   TYPE WordType IS ARRAY (1 .. MaxWordLength) OF Character;

   PROCEDURE GetChar (Char : OUT Character) IS
      -- Input first non-blank character from the input file.
      -- No checks are made for end-of-line or end-of-file.
   BEGIN
      Get(Char);
      WHILE Char = Blank LOOP
         Get(Char);
      END LOOP;
   END GetChar;
```



```
   PROCEDURE PutWord (Word : IN WordType; Length : IN Integer) IS
      -- Output a word of Length characters, stored in Word.
   BEGIN
      FOR I IN 1 .. Length LOOP
         Put(Word(I));
      END LOOP;
   END PutWord;

   PROCEDURE GetWord (Word : OUT WordType; Length : OUT Integer) IS
      -- Inputs a word (stripping off leading blanks),
      -- and stores it in Word.  Sets Length to length of the
      -- word that is read in.  No checks are made for end-of-line
      -- or end-of-file nor for overflow of the array Word.
      Ch : Character;
   BEGIN
      Length := 0;   -- Initialise word length.

      -- Now skip over any leading blanks to find first
      -- character of a word.
      GetChar(Ch);

      -- Now store characters in Word till a blank is reached.
      WHILE Ch /= Blank LOOP
         Length := Length + 1;
         Word(Length) := Ch;       -- Store character in Word array
         Get(Ch);                  -- and then read the next one.
      END LOOP;

   END GetWord;

   PROCEDURE ProcessInitials IS
      -- Input all initials on the line (skipping over any blanks)
      -- and output each initial followed by a full-stop and a blank.
      Initial : Character;
   BEGIN
      WHILE NOT End_Of_Line LOOP
         GetChar(Initial);
         Put(Initial);
         Put(FullStop);
         Put(Blank);
      END LOOP;
   END ProcessInitials;

   Surname    : WordType;   -- To hold a surname,
   NameLength : Integer;    -- and its number of characters.

BEGIN   -- Main program of NameChange.

   OpenInput("oldnames.txt");
   OpenOutput("newnames.txt");

   WHILE NOT End_Of_File LOOP
      -- Input and store the surname
      GetWord(Word => Surname, Length => NameLength);

      -- Input and output the initials.
      ProcessInitials;

      -- Output the (stored) surname.
      PutWord(Word => Surname, Length => NameLength);

      Skip_Line;     -- Move to start of next line in input file.
      New_Line ;     -- Write an e-o-l marker to output file.
   END LOOP;

   CloseInput;
   CloseOutput;

END NameChange;
```