CS141: Lab 6

1. Overview

How To: We'll be dealing with arrays (types that can hold more than one value at a time) and strings (arrays of characters.) You will use arrays to write a program that reads text from the keyboard, keeps a running count of the number of each letter in the text, and capitalizes the sentence.

Prerequisites: *Read these notes carefully first.* This document contains brief introductions to strings and arrays. It will explain how to read a string from the keyboard (even a partial string) files and exceptions. You should refer to previous labs for help with writing programs, input, output, user-defined types, and loops.

Objectives: Understand the following:

- Declaring arrays, including their ranges, types, and initialization.
- Accessing array elements.
- Strings are arrays of characters.
- How to read strings when you don't know how long the string will be.
- Using strings to buffer input and change character data from one form to another.

Why This Is Important: The previous lab introduced the filter and claimed that the skills in labs 1-5 are sufficient for writing many filters. Arrays make a programming language much more powerful, because arbitrarily large amounts of data can be stored inside a program without resorting to files.[1] Arrays and strings are ubiquitous in imperative programming languages, and are usually similar to the arrays in Ada.

Also, arrays are used extensively in the third and fourth homework programs. The techniques used for reading strings in this lab are used in both assignments.

Keep In Mind:

- Arrays contain many values of the same type, like pages in a book.
- The indices into an array are usually a range of numbers, but may also be enumerations or subtypes.
- Strings are arrays of characters; therefore, anything said about arrays applies 100% to strings also.
- Strings can be used to 'buffer input' before turning it into its final form. That means reading into a string, and then turning it into something else. so the string is sometimes *not* the final form of the data. This is a complementary technique to using exceptions to make sure that input data is valid.

2. Assignment and Grading

Write a program that repeatedly reads lines from the keyboard, up to 40 characters long. The program should convert the input to uppercase, replace any non-letter characters with spaces, and print out the new string. You should ignore anything after the 40th character. The program should exit when a blank line is read.

The program should also keep a running total of occurrences of letters A-Z in the input. Count uppercase and lowercase values as being the same letter (ex: 'A' and 'a' are the same.) Print out the final totals before the program exits. It is not necessary to print the totals after every line.

A script with testing examples appears in Section 4.

Grading:

A: Submitted on time and works as described. Sufficiently robust to handle inputs that are too long.

B: Mostly correct capitalization and letter counts.

C: Major problems with letter counts or with long input.

D: Compiles and runs and does meaningful I/O. Good pseudocode in comments.

1 It is provable in theory that an imperative programming language with arrays is at least as powerful as any other programming language, in the sense that they can all be used to solve the same set of problems (which is not to say that it's always easy!)

3. Lab Procedure

3.1 Working With Arrays

Arrays are one of two *compound data types*, which are more complex than ordinary variables. (The other type is *records*, to be dealt with in a later lab.) It is a way to collect many values into a single variable. Arrays are like a book; each page has its own value (which may or may not have anything to do with the other pages), but all the pages are part of the same structure. They may be manipulated individually, or all together, and it is easier than keeping track of a lot of separate variables. There are two components to an array declaration: the *range* of the array, and the type that the array contains.

The array may contain any type, even strings or another array. Each element in the array is one instance of the type, like a page in a book. You access different elements of the array using the parentheses, or the *subscript* notation. The contents of the parentheses are an expression for some place in the array's range. The whole expression has the type of the array's contents, not of the array itself. For example, `ArrayName(10)` means 'The thing in cell 10'. You can either assign into or out of array elements (they can be on either the left or the right side of a `:=` operator.) Some more examples:

```
Type CharArray10 is (1..10) of Character;
Type IntArray10 is (1..10) of Character;
a : CharArray10;
b : IntArray10;
c : Character;
-- Later...
-- Stores 'Z' in element 5
  a(5) := 'Z';
-- Puts 'B' to the screen
  a(1 + 1) := 'B';
  Ada.Text_IO.put(Item => a(2));
-- Prints '8'
  b(5) := 88 / 11; to the screen
  Ada.Integer_Text_IO.Put(Item => b(5));
```

Arrays are usually declared as a Type and then used to declare one or more instances. Arrays may also be used without declaring a type, by directly declaring a variable as an array. However, it's advisable to declare an array as a type if it's going to be used more than once; anonymous types are not necessarily compatible. For example:

```
Type MyArray Is Array (1..10) of Integer;
a1, a2 : MyArray;
a3 : Array (1..10) of Integer;
-- Later ...
a1 := a2; -- Will work
a3 := a2; -- May not work
```

3.2 Arrays with Non-Numerical Ranges

The *range* determines the size of the array. It also indicates what values may be used for the index. Ada is flexible than other languages in that it allows the use of any discrete, finite type as array indices. This feature allows you to easily associate values from one set of symbols with another, as Enumerations do. Unlike enumerations, the associated values (stored in the array) may change. For example, You might use the following to count the letters in the input:

```
SUBTYPE LetterRange IS CHARACTER RANGE 'A'..'Z';
counts: Array (LetterRange) of Integer;
-- Later
counts('I') := counts('I') + 1;
```

```
-- Or...
c := 'I';
counts(c) := counts(c) + 1;
```

3.3 Assigning Every Element of an Array At Once

Since arrays contain more than one value, it is useful to have a syntax that can assign more than one value at a time. Ada provides several options, which resemble the parameter lists used when calling functions and procedures. Array indices are paired with values in a list. The `OTHERS` keyword can be used for 'Anything else that I haven't explicitly named.' It is not possible to assign only part of an array; either every value must be explicitly named, or the `OTHERS` value must be passed. Consider these examples. The final example will raise a constraint error at runtime.

```
SUBTYPE LetterRange IS CHARACTER RANGE 'A'..'Z';
counts: Array (LetterRange) of Integer;
-- Later
counts := (OTHERS => 0); -- Set everything to 0
counts := ('I' => 5, OTHERS => 0); -- Set just I to 5, all else to 0
counts := ('A' => 10, 'E' => 5); -- Wrong
```

The array indices can be omitted if you are assigning all the values in order. This syntax should be used with caution, however, and is best suited to arrays with numerical ranges.

```
counts: Array (1..5) of Integer;
-- Later
counts := (10, 20, 30, 40, 50); -- Set in increments of 10
counts := (10, OTHERS => 0); -- Set first to 10, all else to 0
```

It is also possible to declare and initialize an array (or any variable) at the same time. This:

```
Type MyArray IS 1..5 of Integer;
S1 : MyArray := (10, 20, 30, 40, 50);
```

Does the same thing as this:

```
Type MyArray IS 1..5 of Integer;
S1 : MyArray;
BEGIN
S1 := (10, 20, 30, 40, 50);
```

3.4 Working With Strings

So far, strings have only been used as literals. For example, the following PUT statement uses a string literal for a message: `Ada.Text_IO.Put(Item => "Please input a number >");`

It's important to remember that strings are arrays of characters, so anything that applies to arrays also applies to strings. For example, accessing elements within the string is done the same way as arrays. Also, like arrays, you can only assign literals to strings that are the same length as the string. Some examples are shown below.

```
-- You can declare a six-element string like this ...
  s2: String(1..6);
-- Or like this ...
  MyRange is 1..6;
  s3: String(MyRange);
-- Examples of assigning strings; these two do the same thing.
  s1 := "Dog   ";
  s1 := (1 => 'D', 2 => 'o', 3 => 'g', OTHERS => ' ');
-- Examples of accessing an element of a string
  c1 := s1(3); -- Evaluates to 'g'
```

3.5 Ada.Text_IO.Get_Line

Strings are read and written using the `Ada.Text_IO` package. You are already familiar with the `GET` and `PUT` methods used by this package from other labs. The `Ada.Text_IO.Get_Line` procedure is useful when input is organized into lines. This procedure will read an entire line (up to a newline in a file, or pressing enter on the console) and passes back the subscript of the last character read in `Last`.

An example of `Get_Line` appears below:

```
s1 : String(1..63);
i: Integer;
-- Later, in the body
Ada.Text_IO.Get_line(Item => s1, Last => i);
ada.Text_IO.Put(item => "Number of characters you typed: ");
ada.Integer_Text_IO.Put(item => i);
```

The `Last` parameter is an `OUT` parameter and will store the number of characters read. The read characters are stored into the string. Suppose you typed '`My String`' to the example above. This would set the first nine elements of `s1` to `('M', 'y', ' ', 'S', 't', 'r', 'i', 'n', 'g')`, and `i` would be set to 9. The remaining elements in the string would not be changed. The only way to know how many characters were read is to examine the value passed back in the `Last` parameter.

If you entered a string that was longer than 63 characters, then the first 63 would be stored in `s1` and `i` would be set to 63. The remainder would be discarded.

As you may guess, `Get_Line` can also be used with files, by passing the `File` parameter, just like other Text_IO procedures.

3.6 Buffering Data with Strings

The combination of loops, strings, Get_line, and Skip_line allow for a powerful input buffering mechanism. *Input buffering* refers to storing data when it comes into the program in preparation for some other processing. In this section we describe the general process of using strings to buffer input data. Specific code is left as an exercise to the student and will often be specific to the program.

A very common buffering operation is input validation, which means making sure that the input matches what the program expects. Input validation is complementary to exceptions. You can mix and match buffering and exceptions as appropriate. For example, you might buffer lines from a file in a string, and use exceptions to detect errors opening the file.

It's an interesting quirk of `Get_Line` that it does not report any error if more than the size of the array was available. Therefore, in order to tell if the user entered 'too much' data, buffers for `Get_Line` should be declared as one longer than the expected size of the input. The program can use a read of exactly the length of the string as an indicator to `Skip_Line`.

The general procedure goes something like this:

1. Declare a string type for your buffer. This will usually be slightly longer than the length of the longest record you expect to read, or the longest line if you are using line-oriented I/O.
2. Read the string from the file or console using `Get_line`.
3. If more characters were read than the longest expected record, then the line was too long, and the read pointer will be sitting somewhere in the middle of the line. You may use `Skip_Line` to discard the rest and ensure that the pointer is at the start of a record.
4. Check the input. Often this will mean using a loop to step over every character in the string, possibly converting it to another type before storing it somewhere else (like in another array.)
5. If the input passed, continue with the program. If the input failed, discard the input and go to 2.

3.7 Important Hint about Changing Uppercase to Lowercase

You can use the `'Pos` and `'Val` attributes on Character and subtypes of Character to get the position of a letter within a character set (UNICODE or ASCII, depending on the computer.)[2] That means that you can convert lowercase to uppercase by finding the distance of a letter from `A`, and adding that much to the position of `a`. The following expression will convert the character stored in c from lowercase to uppercase.

```
Character'Val(Character'Pos(c) + Character'Pos('A') -
Character'Pos('a'));
```

A similar technique will be used in homework 3 to turn a character into a decimal digit.

Alternatively, the Ada library includes a function for turning characters into uppercase.

```
With Ada.Character.Handling;
c : Character;
-- Later, in body...
c := Ada.Characters.Handling.to_upper(c);
```

3.8 Testing membership in an Enumeration or Range

The `IN` operator can be used to test if a value is a member of a type or subtype and will return `TRUE` or `FALSE`. The following code fragments show examples of the `IN` operator.

```
SUBTYPE SmallInt IS Integer Range 1..100;
TYPE Colors IS (Red, Green, Yellow, Blue);
SUBTYPE Uppercase IS Character Range 'A'..'Z';
TYPE LetterGrades IS ('A', 'B', 'C', 'D', 'F');
a: SmallInt := 5;
c: Colors := Green;
-- The following are EXPRESSIONS and not entire statements.
a in SmallInt       -- will be TRUE
1000 in SmallInt    -- will be FALSE
c in Colors         -- will be TRUE
'D' in LetterGrades -- will be TRUE
'Z' in letterGrades -- Will be FALSE
```

4. Testing and Submitting the Program

Run the program. An example script appears below. You are welcome to make up your own test and you need not match the prompts exactly.

```
> xyzzyx
XYZZYX
> A man, a plan, a canal, panama.
A MAN  A PLAN  A CANAL  PANAMA
> The quick brown fox jumps over the lazy dog.
THE QUICK BROWN FOX JUMPS OVER THE LAZY
>
Counts:
A 11
B 1
C 2
D 0
...
```

2 Technically speaking, there are other character sets (such as EBCDIC, common on old IBM mainframes) where the letters are not in order. However, those situations are uncommon (and in the EBCDIC case this trick still works.)