Chapter 11

Handling Strings

11.1 Length of a String

In this chapter we will look at ways that you can work with strings. All the different operations that you need to perform can be accomplished by two basic string operations: joining two strings together and selecting a part of a string. As well, you need to use the predefined function length to tell you the number of characters in a string.

It is easy to find the length of any string stored in a variable using the Turing predefined function length.

Here is a program that outputs the length of words.

```
% The "WordLengths" program
% Read words and find their length
put "Enter one word to line, end with 'end' "
var word : string
loop
    put "Enter word: " ..
    get word
    exit when word = "end"
    put word, " has ", length (word), " letters"
end loop
```

Here is a sample Execution window.

```
Enter one word to line, end with 'end'
Enter word banana
banana has 6 letters
Enter word kangaroo
kangaroo has 8 letters
Enter word end
```

11.2 Joining Strings Together

The operation of joining strings together is called **catenation** and is accomplished using the operator +. When this operator is between two numbers it means that they are to be added. When the + operator is between two strings it means they are to be joined. Here is an example.

```
put "O" + "K"
```

It produces the output *OK*; the two strings are joined.

Here is a program that reads five words and joins them into a line of output.

```
% The "WordsOnLine" program
% Reads 5 words and outputs all on a line
var word : string
var line := ""    % Initialize line to the empty string
put "Enter 5 words"
for i : 1 .. 5
    get word
    line := line + word + " "
end for
put line
```

Here is a sample Execution window.

```
Enter 5 words
Why am I entering this many words
Why am I entering this
```

If the words were entered one to a line the output would be given right after you enter the 5th word and the window would look like this

```
Why
am
I
entering
```

```
this
Why am I entering this
```

Notice that in the declaration of the string variable *line* it is initialized to the empty string. In the **for** loop the most recently read *word* is catenated onto the line and a blank is catenated on to that; the result is stored in the variable *line*. If the blank were not catenated the output would be

```
WhyamIenteringthis
```

11.3 Selecting Part of a String

A part of a string is called a **substring**. We define a substring by placing in parentheses after the name of the string: the position of the first character in the substring, then two dots, followed by the position of the last character in the substring. The character positions of a string are numbered starting on the left from 1 to the length of the string. The output for this program

```
const word := "magnet"
put word (4 .. 6)
```

would be *net*. The substring *net* consists of the 4th to the 6th characters of *magnet*.

If the substring goes from some position to the end of the string then the last position can be written as an asterisk. So the program we just had would give the same result if the **put** statement were

```
put word (4 .. *)
```

Here is a program to read a series of words and output the last three characters of each word until the word *quit* is read.

```
% The "FinalThreeLetters" program
% Read words and if possible give last three characters
var word : string
```

```
put "Enter words one to a line, end with 'quit' "
loop
    put "Enter word: " ..
    get word
    exit when word = "quit"
    if length (word) >= 3 then
        put word (* – 2 .. *)
    else
        put "Word has fewer than 3 characters"
    end if
end loop
```

Notice that the third last character can be given as ** – 2* in the substring specification.

You must not ask for a character position that is not present, for example, a character position whose number is less than 1 or greater than the length of the word or an execution time error will result. In general, the first character position's value must be between 1 and the last character position's value inclusive. In the particular case where it is one greater than the last character position the result is an empty string. For example, *word (*length *(word)+ 1 .. **) gives the empty string.

Here is a sample Execution window for the *FinalThreeLetters* program.

```
Enter words one to a line, end with 'quit'
Enter word speaking
ing
Enter word softly
tly
Enter word to
Word has fewer than 3 characters
Enter word quit
```

If we had not tested in the **if** statement to see if the *word* was at least 3 characters long and had asked for the last three characters of *to* we would have been told of an execution error. You should try this program and see what you get

```
const word := "to"
```

```
put word (* – 2 .. *)
```

There will be an execution error because * – 2 attempts to locate a character before the first character.

If the substring is to be a single character you can just put that character's position in parentheses; a range is not required.

Here is a program that outputs each letter of a word on a separate line.

```
% The "LetterAtATime" program
% Read a word and output it a letter-at-a-time
var word : string
put "Enter words, end with 'tired' "
loop
    put "Enter word: " ..
    get word
    exit when word = "tired"
    for i : 1 .. length (word)
        put word (i)
    end for
end loop
put "Why not take a rest?"
```

Here is a sample Execution window.

```
Enter words, end with 'tired'
Enter word sick
s
i
c
k
Enter word tired
Why not take a rest?
```

11.4 Searching for a Pattern in a String

We have seen several examples where we recognized a word such as *stop* or *quit* that we have read. We also need to be able to recognize if a certain pattern of characters is in a string or not. For example, we might want to look for words containing the pattern ÒieÓ. To do this we use the predefined Turing function index whose value is the character position where a pattern first matches a string. The value of

index (*string*, *pattern*)

is the first position from the left in *string* where the *pattern* matches. For example,

index ("getting", "t")

has a value 3. There is a second occurrence of *t* in *getting* at position 4. The value of

index ("dandelion", "lion")

is 6. If there is no match at all the value of index is zero.

Here is a program to detect words that contain the letter *s*.

```
% The "HaveAnS" program
% Test to see if a word contains an "s"
var word : string
put "Enter a series of words, end with 'last' "
loop
    put "Enter word: " ..
    get word
    exit when word = "last"
    if index (word, "s") not= 0 then
        put word, " contains an 's' "
    else
        put word, " does not contain an 's' "
    end if
end loop
```

Here is a similar program to see if a word contains two occurrences of a pattern. It is a more difficult process. This time the program will ask you to enter the pattern you want to test for.

```
% The "DoublePattern" program
% Test to see if a word has two occurrences of a pattern
var word, pattern : string
put "Enter the pattern you want to test for: " ..
get pattern
const size := length (pattern)
put "Enter a series of words"
put "Enter 'finis' to stop"
loop
    get word
    exit when word = "finis"
    const place := index (word, pattern)
    if place = 0 then
        put word, " contains no occurrences of ", pattern
    elsif index (word (place + size .. *), pattern) = 0 then
        put word, " contains one occurrence of ", pattern
    else
        put word, " contains at least two occurrences of ", pattern
    end if
end loop
```

In this program if you find that the word contains one occurrence of the pattern then you must see if there is a second occurrence. The second search for the pattern must be in the substring of *word* starting after the end of the first occurrence and going to the end.

To see whether or not you have the correct expressions for the conditions try working through a test case or two on paper rather than on the computer. For example, suppose the pattern is ÒsÓ and the word is ÒwallsÓ. The value of *size* is 1 and *place* is 5. The value of *place* + *size* is 6. The value of the substring *word (place* + *size .. *)* will be the empty string which is what you get if the value of the beginning of the substring range is one greater than the length of the string which is 5. Now try the word ÒglassÓ. The value of *place* will be 4 and *place* + *size* will be 5.

The substring *word (place+size .. *)* is then really *word (5 .. 5)* which is just the last character.

11.4.1 Counting Patterns in a Word

The previous program shows how to count the patterns in a word. If the word is ÒbananaÓ and the pattern is ÒaÓ then the program will tell you that there are at least two occurrences of ÒaÓ in ÒbananaÓ. It is also possible to modify the program so that the exact count will be outputt.

Here is the revised program that inputs the pattern and outputs the count.

```
% The "DoublePattern2" program.
var pos, count, size : int
count := 0
var word, pattern : string
put "Enter the pattern that you want to search for: " ..
get pattern
put "Enter the word you want to search through: " ..
get word
size := length (pattern)
loop
    pos := index (word, pattern)
    exit when pos = 0
    count := count + 1
    word := word (pos + size .. *)
end loop
put "The number of occurrences: ", count
```

You can see how *count* changes by tracing through the program. The following trace shows how the *DoublePattern2* program runs when *pattern* is set to ÒaÓ and *word* is set to ÒbananaÓ.

```
count=0
size=1
first time through loop
    pos=2
    count=1
```

```
     word=nana
second time through loop
     pos=2
     count=2
     word=na
third time through loop
     pos=2
     count=3
     word=""
fourth time through loop
     pos=0
loop exited and the count of 3 is output
```

11.5 Substituting One Pattern for Another

We showed examples where we looked for a pattern in a word. We could have substituted a different pattern after we found the one we were searching for. This would then be a search and substitute process. For example, in the program *HaveAnS*, in the previous section we could have substituted another letter such as ÒtÓ for the ÒsÓ.

In Turing, you cannot assign to a substring. In order to change part of a string, you must rebuild the string. For example, to change the letter ÒsÓ to ÒtÓ in a string, you would create a new string by concatenating the part of the string up to (but not including) the ÒsÓ with the ÒtÓ and then concatenating the result with the part of the string from just past the ÒsÓ to the end of the string.

Here is the assignment statement that would do it. Place this statement after the **put** in the **then** clause

```
word := word (1 .. index (word, "s") – 1) + "t" +
               word (index (word, "s") + 1 .. *)
```

The new *word* is made up of three pieces catenated. The first piece is the substring of the original *word* up to the position of the pattern. Next comes the substituted letter ÒtÓ, then the substring of the original *word* starting after the pattern and going to the end. If the pattern is already at the end of the word this last substring will be a empty string.

Sometimes in programs the same function is evaluated several times. It is often more efficient to assign its value to a variable, then use the variable instead of reevaluating the function. For example, in the *twice* program we assigned to the variable *place* the value of index *(word, pattern)*.

11.6 Eliminating Characters from Strings

Sometimes it is useful to be able to eliminate a certain class of characters from a string. Here is a program which removes all the vowels from a word.

```
% The "RemoveVowels" program
% Eliminates the vowels from a word
var word : string
put "Enter a series of words, end with '*' "
const vowels := "aeiou"
loop
    get word
    exit when word = "*"
    var newWord := ""       % empty string
    for i : 1 .. length (word )
        if index (vowels, word (i )) = 0 then
            % Letter is not a vowel
            newWord := newWord + word (i )
        end if
    end for
    put "Word without vowels ", newWord
end loop
```

In the **for** loop each letter of *word*, namely *word (i)*, is tested as the pattern against the string of vowels. If it is found in the string of vowels the *index* function will not be zero and we do not catenate it onto the new word we are forming.

Here is a sample Execution window with the keyboard input in bold.

```
Enter a series of words, end with '*'
jump
Word without vowels jmp
diagonal
Word without vowels dgnl
*
```

We can also remove a pattern from within a string using *word* by

word := *word* (1 .. *pos* – 1) + *word* (*pos* + *size* .. *)

where *pos* is the position of the pattern in the string and *size* is the size of the pattern. This redefines *word* so that it contained the part of the word before the pattern (*word* (1 .. *pos* – 1)) and the part of the word after the pattern (*word* (*pos* + *size* .. *)).

Here is a modification of the *DoublePattern2* program that outputs the string with all occurrences of a pattern removed.

```
% The "DeletePattern" program.
var pos, size : int
var word, pattern : string
put "Enter the pattern that you want to search for: " ..
get pattern
put "Enter the word you want to search through: " ..
get word
size := length (pattern)
loop
    pos := index (word, pattern)
    exit when pos = 0
    word := word (1 .. pos – 1) + word (pos + size .. *)
end loop
put "Here is the word with the pattern removed: ", word
```

11.7 Bullet-Proofing Programs

Another important consideration when designing programs is making sure that the user cannot crash the program by entering unexpected data. This is called Òbullet-proofingÓ your program. The most common kind of error that can occur is if the user enters a letter of the alphabet as input when asked for a number. When this happens, Turing is unable to read the input as a number and the program stops execution and outputs an error message. It creates a run-time error.

Reading all input as strings avoids this problem. The string is converted to an integer or real only after making certain that the string contains valid input. If the string does not contain valid input, the program can ask the user to reenter a proper value.

Here is an example of a program segment that gets an integer value from the user.

```
var input : string
var age : int
put "Enter your age: " ..
loop
    get input
    exit when strintok (input)
    put "Not a number. Please enter an integer: " ..
end loop
age := strint (input)
```

The program prompts the user for input and then enters the loop. In the loop, the user inputs their age into the string variable *input*. The program then checks whether *input* can be converted to an integer. The strintok (pronounced *strint-okay*) built-in subprogram examines *input* and returns true if *input* can be converted to an integer and false otherwise. If *input* cannot be converted to an integer, the program outputs an error message and loops back, asking for the another input. If the user enters valid input the program leaves the loop. The string is then converted into an integer using the strint function.

A similar program segment for reading in a real number can be created using the strrealok and strreal perdefined subprograms.

11.8 Exercises

1. Write a program to count the total number of characters in a series of 10 words that you enter, and compute the average word length.
2. Write a program to output the first and last letters of a series of words. A sample Execution window might be:

```
Enter a series of words one to a line, end with
'wow'
Enter word pig
pg
Enter word dog
dg
Enter word a
Word has only 1 character
Enter word wow
```

3. Write a program which produces a line of asterisks of a given length by catenating enough single asterisks in a **for** loop. Here is a sample Execution window:

```
Enter a negative number to stop
How many asterisks do you want? 8
********
How many asterisks do you want? 5
*****
How many asterisks do you want? -1
```

The *repeat* predefined function can be used to do this too. For example,

```
put repeat ("*", 5)
```

will result in the output *****. Patterns of more than one character can be repeated also, for example, repeat ("Hi", 3) produces three ÒHiÓs.

4. Rewrite the program of question 3 so that the pattern to be repeated by catenation is read into the computer. Try several patterns.
5. Write a program to change words made up of lower case letters into a secret code. The letter *a* is to be changed to *b*, *b* to *c*, and so on; *z* becomes *a*. To do this you must know a little about the ASCII code shown in the appendix. The predefined function ord has a value equal to the ASCII code of the letter which is its parameter. For example, ord ("a") has the value 97 which is the ASCII equivalent of the letter *a*. The letter *b* has the code 98. The predefined function chr can translate back from a value to a letter. For example, the function chr (97) has a value the character *a*. To change an *a* to a *b* you would use both functions, one after the other. The value of

   ```
   chr (ord ("a") + 1)
   ```

   is *b*. In this way you can convert to the secret code. Try your luck at this.
6. Write a program to read a series of words from the keyboard and output the reverse word with the letters backward. Keep all the letters of the word in the same case: upper or lower. Here is a sample window.

   ```
   Enter a word COW
   The reverse word is WOC
   Enter a word madam
   The reverse word is madam
   (etc.)
   ```

   If the reverse word is the same as the word, the word is a **palindrome**. If you find this to be the case output a line saying

   ```
   This is a palindrome
   ```
7. Read a series of words and output the middle letter of each word that has an odd number of letters or announce that the

word has an even number of letters. Here is a sample Execution window:

```
Enter word brine
The middle letter is i
Enter word bright
The word has an even number of letters
(etc.)
```

Use the end-of-file signal to stop the repetition. Try putting the input on a disk file called *list* and redirect the input to be from it. Does your output look the same as before? How could you change your program so that you see the word that is read.

8. Write a program which gives the user this menu.

```
Menu
1. Count a pattern
2. Eliminate a pattern
3. Substitute a pattern
4. Exit
```

If the user chooses #1 - ask for a word and a single letter pattern. Display the number of times the pattern occurs in the word. (banana, a, 3)

If the user chooses #2 - ask for a word and a single letter pattern. Display the word without the pattern (banana, a, bnn)

If the user chooses #3 - ask for a word, a single letter search pattern and a single letter replacement pattern. Display the word with the alterations. (banana, a, o, bonono)

If the user chooses #4 - quit the program

Use getch and clear the Execution window between options.

9. Modify alternative 3 in Exercise 8 so that it properly handles the case of replacing each o in moon by oo.

10. Using the *RemoveVowels* program as your guide, write a program which inputs a string and then outputs each character in the string and whether or not the character is a vowel, a consonant, a number or any other character. For example if Ò5te+Ó were input, then the output should be:

```
5 is a number
```

```
t is a consonant
e is a vowel
+ is any other character
```

11. Modify Exercise 10 so that it outputs the vowels, consonants, numbers, and other characters as words. For example if Òasdfert456u2~1?Ó were input then the output would be:

```
numbers - 45621
vowels - aeu
consonants - sdfrt
any other character - ~?
```

12. Use your ingenuity to come up with a different way of finding out whether or not a word is a palindrome.
13. Write a program that inputs a word, a letter, and a replacement letter. If the first letter exists in the word, all occurrences of it should be replaced by the replacement letter and the new ÒwordÓ printed . If the letter does not exist in the word, the message Òno replacement neededÓ should be printed. For example:

```
Enter a word: program
Enter a letter: r
Enter replacement letter: l
New “word” is plogram.
```

14. Repeat Exercise 8 but use patterns of more than 1 letter (replace a pattern of letters in a word with another pattern of letters).

11.9 Technical Terms

length of string
catenation
substring
range of substring
index function
searching for pattern
substitution of one string by another
deletion of characters
insertion of characters
repeat function
ord function
chr function
palindrome