Chapter 14
Characters

These metaphysics of magicians, And necromantic books are heavenly; Lines, circles, letters and characters.
Christopher Marlowe, The Tragical History of Doctor Faustus

Aims
The aims of this chapter are:

- To extend the ideas about characters introduced in earlier chapters.
- To demonstrate that this enables us to solve a whole new range of problems in a satisfactory way.

14.1 Introduction

For each type in a programming language there are the following concepts:

- Values are drawn from a finite domain.
- There are a restricted number of operations defined for each type.

For the character data type the basic unit is an individual character The complete Fortran character set is given in Sect. 4.8 in Chap. 4. This provides us with 95 printing characters. Other characters may be available. The Wikipedia entry

```
http://en.wikipedia.org/wiki/Character_encoding
```

has quite detailed information on how complex this area actually is.

As the most common current internal representation for the character data type uses 8 bits this should provide access to 256 characters. However, there is little

I. Chivers and J. Sleightholme, *Introduction to Programming with Fortran*,
https://doi.org/10.1007/978-3-319-75502-1_14

agreement over the encoding of these 256 possible characters, and the best you can normally assume is access to the ASCII character set, which is given in Chap. 4. One of the problems at the end of this chapter looks at determining what characters one has available.

The only operations defined are concatenation (joining character strings together) and comparison.

We will look into the area of character sets in more depth later in this chapter.

We can declare our character variables:

```
character :: a, string, line
```

Note that there is no default typing of the character variable (unlike integer and real data types), and we can use any convenient name within the normal Fortran conventions. In the declaration above, each character variable would have been permitted to store one character. This is limiting, and, to allow character strings which are several units long, we have to add one item of information:

```
character (10) :: a
character (16) :: string
character (80) :: line
```

This indicates that `a` holds 10 characters, `string` holds 16, and `line` holds 80. if all the character variables in a single declaration contain the same number of characters, we can abbreviate the declaration to

```
character(80) :: list, string, line
```

But we cannot mix both forms in the one declaration. We can now assign data to these variables, as follows:

```
a='first one '
string='a longer one'
line='the quick brown fox jumps over the lazy dog'
```

The delimiter apostrophe (') or quotation mark (") is needed to indicate that this is a character string (otherwise the assignments would have looked like invalid variable names).

14.2 Character Input

In an earlier chapter we saw how we could use the `read *` and `print *` statements to do both numeric and character input and output or I/O. When we use this form of the statement we have to include any characters we type within delimiters (either the apostrophe ' or the quotation mark “). This is a little restricting and there is a slightly more complex form of the `read` statement that allows one to just type the string on its own.

*14.2.1 Example 1: The * Edit Descriptor*

The following two programs illustrate the differences:

```
program ch1401
!
! Simple character i/o
!
  character (80) :: line

  read *, line
  print *, line
end program ch1401
```

This form requires enclosing the string with delimiters.

14.2.2 Example 2: The a Edit Descriptor

Consider the next form:

```
program ch1402
!
! Simple character i/o
!
  character (80) :: line

  read '(a)', line
  print *, line
end program ch1402
```

With this form one can just type the string in and input terminates with the carriage return key. The additional syntax `'(a)'` where `'(a)'` is a character edit descriptor. The simple examples we have used so far have used implied format specifiers and edit descriptors. For each data type we have one or more edit descriptors to choose from. For the character data type only the `a` edit descriptor is available.

14.3 Character Operators

The first manipulator is a new operator — the concatenation operator //. With this operator we can join two character variables to form a third, as in

```
character (5) :: first, second
character (10) :: third
first='three'
second='blind'
...
third=first//second
.
third=first//'mice'
```

where there is a discrepancy between the created length of the concatenated string and the declared lengths of the character strings, truncation will occur. For example,

```
  third=first//' blind mice'
```

will only append the first five characters of the string ‘blind mice’ i.e., ‘blin’, and third will therefore contain ‘three blin’.

What would happen if we assigned a character variable of length ‘n’ a string which was shorter than n? For example,

```
character (4) :: c2
c2='ab'
```

The remaining two characters are considered to be blank, that is, it is equivalent to saying

```
c2='ab  '
```

However, while the strings ‘ab’ and ’ab ‘are equivalent, ‘ab’ and ‘ab’ are not. In the jargon, the character strings are always left justified, and the unset characters are trailing blanks.

If we concatenate strings which have ‘trailing blanks’, the blanks, or spaces, are considered to be legitimate characters, and the concatenation begins after the end of the first string. Thus

```
character (4) :: c2,c3
character (8) :: jj
c2='a'
c3='man'
jj=c2//c3
```

```
print*, 'the concatenation of ',c2,' and ',c3,' is'
print*,jj
```

would appear as

```
the concatenation of a   man gives
a   man
```

at the terminal.

14.4 Character Substrings

Sometimes we need to be able to extract parts of character variables — substrings. The actual notation for doing this is a little strange at first, but it is very powerful. To extract a substring we must provide two items:

- The position in the string at which the substring begins.
- The position at which it ends.

In the examples that follow we will use the following

```
string='share and enjoy'

Substring                 Characters
string(3:3 )              a
string(3:5 )              are
string(:3 )              sha
string(11: )              enjoy
```

Character variables may also form arrays:

```
character (10) , dimension(20) :: a
```

sets up a character array of twenty elements, where each element contains ten characters. In order to extract substrings from these array elements, we need to know where the array reference and the substring reference are placed. The array reference comes first, so that

```
do i=1,20
  first=a(i)(1:1)
end do
```

places the first character of each element of the array into the variable first. The syntax is therefore ‘position in array, followed by position within string’.

Any argument can be replaced by an integer variable or expression:

```
string(i:j)
```

14.4.1 Example 3: Stripping Blanks from a String

This offers interesting possibilities, since we can, for example, strip blanks out of a string:

```
program ch1403
  implicit none
  character (80) :: string, strip
  integer :: ipos, i, length = 80

  ipos = 0
  print *, ' type in a string'
  read '(a)', string
  do i = 1, length
    if (string(i:i)/=' ') then
      ipos = ipos + 1
      strip(ipos:ipos) = string(i:i)
    end if
  end do
  print *, string
  print *, strip
end program ch1403
```

14.5 Character Functions

There are special functions available for use with character variables: `index` will give the starting position of a string within another string.

14.5.1 Example 4: The `index` Character Function

If , for example, we were looking for all occurrences of the string ‘Geology’ in a file, we could construct something like:

```
program ch1404
  implicit none
  character (80) :: line
  integer :: i

  do
    read '(a)', line
    i = index(line, 'Geology')
    if (i/=0) then
      print *, &
        ' String Geology found at position ', i
      print *, ' in line ', line
      exit
    end if
  end do
end program ch1404
```

There are two things to note about this program. Firstly the `index` function will only report the first occurrence of the string in the line; any later occurrences in any particular line will go unnoticed, unless you account for them in some way. Secondly, if the string does not occur, the result of the `index` function is zero, and given the infinite loop (`do  enddo`) the program will crash at run time with an end of file error message. This isn't good programming practice.

14.5.2 The `len` and `len_trim` Functions

The `len` function provides the length of a character string. This function is not immediately useful, since you really ought to know how many characters there are in the string. However, as later examples will show, there are some cases where it can be useful. Remember that trailing blanks do count as part of the character string, and contribute to the length.

14.5.3 Example 5: Using `len` and `len_trim`

The following example illustrates the use of both `len` and `len_trim`:

```
program ch1405
  implicit none
  character (len=20) :: name
  integer :: name_length
```

```
    print *, ' type in your name'
    read '(a)', name
  ! show len first
    name_length = len(name)
    print *, ' name length is ', name_length
    print *, ' ', name(1:name_length), &
      '<-end is here'
    name_length = len_trim(name)
    print *, ' name length is ', name_length
    print *, ' ', name(1:name_length), &
      '<-end is here'
  end program ch1405
```

14.6 Collating Sequence

The next group of functions need to be considered together. They revolve around the concept of a collating sequence. In other words, each character used in Fortran is ordered as a list and given a corresponding weight. No two weights are equal. Although Fortran has only 63 defined characters, the machine you use will generally have more; 95 printing characters is a typical minimum number. On this type of machine the weights would vary from 0 to 94. There is a defined collating sequence, the ASCII sequence, which is likely to be the default. The parts of the collating sequence which are of most interest are fairly standard throughout all collating sequences.

In general, we are interested in the numerals (0–9), the alphabetic characters (A–Z, a-z) and a few odds and ends like the arithmetic operators (+ – / *), some punctuation (. and ,) and perhaps the prime ('). As you might expect, 0–9 carry successively higher weights (though not the weights 0 to 9), as do A to Z and a to z. The other odds and ends are a little more problematic, but we can find out the weights through the function `ichar`. This function takes a single character as argument and returns an integer value. The ASCII weights for the alphanumerics are as follows:

```
0--9   48--57
A--Z   65--90
```

One of the exercises is to determine the weights for other characters. The reverse of this procedure is to determine the character from its weighting, which can be achieved through the function `char`. `char` takes an integer argument and returns a single character. Using the ASCII collating sequence, the alphabet would be generated from

```
do i=65,90
  print*,char(i)
enddo
```

This idea of a weighting can then be used in four other functions:

function	Action
lle	lexically less than or equal to
lge	lexically greater than or equal to
lgt	lexically greater than
llt	lexically less than

In the sequence we have seen before, A is lexically less than B, i.e., its weight is less. Clearly, we can use `ichar` and get the same result. For example,

```
if(lgt('a','b')) then
```

is equivalent to

```
if(ichar('a') > ichar('b')) then
```

but these functions can take character string arguments of any length. They are not restricted to single characters.

These functions provide very powerful tools for the manipulation of characters, and open up wide areas of non-numerical computing through Fortran. Text formatting and word processing applications may now be tackled (conveniently ignoring the fact that lower-case characters may not be available).

There are many problems that require the use of character variables. These range from the ability to provide simple titles on reports, or graphical output, to the provision of a natural language interface to one of your programs, i.e., the provision of an English-like command language. Software Tools by Kernighan and Plauger contains many interesting uses of characters in Fortran.

14.7 Example 6: Finding Out About the Character Set Available

The following program prints out the characters between 32 and 127.

```
program ch1406
  implicit none
  integer :: i
  do i = 32, 62
    print *, i, char(i), i + 32, char(i+32), &
      i + 64, char(i+64)
  end do
  i = 63
  print *, i, char(i), i + 32, char(i+32), &
    i + 64, 'del'
end program ch1406
```

This is the output from the Intel compiler under Windows.

```
          32                64 @               96 `
          33 !              65 A               97 a
          34 "              66 B               98 b
          35 #              67 C               99 c
          36 $              68 D              100 d
          37 %              69 E              101 e
          38 &              70 F              102 f
          39 '              71 G              103 g
          40 (              72 H              104 h
          41 )              73 I              105 I
          42 *              74 J              106 j
          43 +              75 K              107 k
          44 ,              76 L              108 l
          45 -              77 M              109 m
          46 .              78 N              110 n
          47 /              79 O              111 o
          48 0              80 P              112 p
          49 1              81 Q              113 q
          50 2              82 R              114 r
          51 3              83 S              115 s
          52 4              84 T              116 t
          53 5              85 U              117 u
          54 6              86 V              118 v
          55 7              87 W              119 w
          56 8              88 X              120 x
          57 9              89 Y              121 y
          58 :              90 Z              122 z
          59 ;              91 [              123 {
          60 <              92 \              124 |
          61 =              93 ]              125 }
          62 >              94 ^              126 ~
          63 ?              95 _              127 del
```

Try this program out on the system you use. Do the character sets match?

14.8 The scan Function

The scan functions scans a string for characters from a set of characters. The syntax is given below.

- scan(string,set) - Scans a string for any one of the characters in a set of characters.

14.8.1 Example 7: Using the scan Function

```
program ch1407
  implicit none
  character (1024) :: string01
  character (1) :: set = ' '
  integer :: i
  integer :: l
  integer :: start, end

  string01 = 'The important issue about &
    &a language, is not so'
  string01 = trim(string01) // ' ' // 'much &
    &what features the language possesses, &
    &but'
  string01 = trim(string01) // ' ' // 'the &
    &features it does possess, are sufficient, &
    &to'
  string01 = trim(string01) // ' ' // 'support &
    &the desired programming styles, in &
    &the'
  string01 = trim(string01) // ' ' // &
    'desired application areas.'
  l = len(trim(string01))
  print *, ' Length of string is = ', l
  print *, ' String is'
  print *, trim(string01)
  start = 1
  end = l
  print *, ' Blanks at positions '
  do
    i = scan(string01(start:end), set)
    start = start + i
    if (i==0) exit
    write (*, 100, advance='no') start - 1
  end do
100 format (i5)
end program ch1407
```

Note the use of the `trim` function when using the concatenation operator to initialise the string to the text we want.

The output from one compiler is given below. The text has been wrapped to fit the page

```
Length of string is =          217
String is
The important issue about a language, is not so much
what features the language   possesses,
but the features it does possess, are sufficient,
to support the desired programming styles,
in the desired application areas.
Blanks at positions
  4   14   20   26   28   38   41   45   48   53  58
 67   71   80   91   95   99  108  111  116  125 129
141  144  152  156  164  176  184  187  191  199 211
```

The text in this program is used in two problems at the end of this chapter.

14.9 Summary

Characters represent a different data type to any other in Fortran, and as a consequence there is a restricted range of operations which may be carried out on them.

A character variable has a length which must be assigned in a character declaration statement.

Character strings are delimited by apostrophes (’) or quotation marks (“). Within a character string, the blank is a significant character.

Character strings may be joined together (concatenated) with the // operator.

Substrings occurring within character strings may be also be manipulated.

Table 14.1 has details of a number of functions especially for use with characters.

Table 14.1 String functions in Fortran

Function name	Explanation
achar	Return the character in the ASCII character set
adjustl	Adjust left, remove leading blanks, add trailing blanks
adjustr	Adjust right,remove trailing blanks, insert leading blanks
char	Return the character in the processor collating sequence
iachar	As above but in the ASCII character set
index	Locate one string in another
len	Character length including trailing blanks
len_trim	Character length without the trailing blanks
lle	Lexically less than or equal to
lge	Lexically greater than or equal to
lgt	Lexically greater than
llt	Lexically less than
repeat	Concatenate several copies of a string
scan	Scans a string for anyone of the characters in the set
trim	Remove the trailing blanks
verify	Verify that a set of characters contains all the characters in a string

A detailed explanation is given in appendix D.

14.10 Problems

14.1 Suggest some circumstances where PRIME="" might be useful. What other alternative is there and why do you think we use that instead?

14.2 Write a program to write out the weights for the Fortran character set. Modify this program to print out the weights of the complete implementation defined character set for your version of Fortran. Is it ASCII? if not, how does it differ?

14.3 Write a program that produces the following output.

```
!
"\#
$\%&
```

```
’()*
+,-./
012345
6789:;<<
=>>?@ABCD
EFGHIJKLM
NOPQRSTUVW
XYZ[\]^\_‘ab
cdefghijklmn
opqrstuvwxyz\{
|\}\~
```

We assume the ASCII character set in this example.

14.4 Modify the above program to produce the following output.

```
           !
         "\#$
        \%&’()
       *+,-./0
      123456789
     :;<>?@ABCD
    EFGHIJKLMNOPQ
   RSTUVWXYZ[\]^\_‘
  abcdefghijklmnopq
 rstuvwxyz\{|\}\~
```

Again we assume the ASCII character set.

14.5 Modify program ch1407 to break the text into phrases, using the comma and full stop as breaking characters. The output expected is given below.

```
The important issue about a language
is not so much what features the language possesses
but the features it does possess
are sufficient
to support the desired programming styles
in the desired application areas
```

Modify the above to break the text into words and count the frequency of occurrence of words by length. The output should be similar to that given below.

```
1   a                                              1
2   is so it to in                                 5
3   The not the but the are the the                8
```

```
4   much what does                        3
5   issue about areas                     3
6   styles                                1
7   possess support desired desired       4
8   language features language features   4
9   important possesses                   2
10  sufficient                            1
11  programming application               2
```

14.6 Use the `index` function in order to find the location of all the strings ‘is’ in the following data:

If a programmer is found to be indispensable, the best thing to do is to get rid of him as quickly as possible.

14.7 Find the ‘middle’ character in the following strings. Do you include blanks as characters? What about punctuation?

Practice is the best of all instructors. experience is a dear teacher, but fools will learn at no other.

14.8 In English, the order of occurrence of the letters, from most frequent to least is

```
E, T, A, O, N, R, I, S, H, D, L,
F, C, M, U, G, Y, P, W, B, V, K,
X, J, Q, Z
```

Use this information to examine the two files given in appendix E (one is a translation of the other) to see if this is true for these two extracts of text. The second text is in medieval Latin (c. 1320). Note that a fair amount of compression has been achieved by expressing the passage in Latin rather than modern English. Does this provide a possible model for information compression?

14.9 A very common cypher is the substitution cypher, where, for example, every letter A is replaced by (say) an M, every B is replaced by (say) a Y, and so on. These enciphered messages can be broken by reference to the frequency of occurrence of the letters (given in the previous question).

Since we know that (in English) E is the most commonly occurring letter, we can assume that the most commonly occurring letter in the enciphered message represents an E; we then repeat the process for the next most common and so on. Of course, these correspondences may not be exact, since the message may not be long enough to develop the frequencies fully.

However, it may provide sufficient information to break the cypher.

The file given in appendix E contains an encoded message. Break it.

Clue — Pg +Fybdujuvef jo Tdjfodf, Jorge Luis Borges.

14.10 Write a program that counts the total number of vowels in a sentence or text. Output the frequency of occurrence of each vowel.