Caesar Cipher Program

Important: The method shown for this algorithm is *not* suitable to adapt for the week eight assignment. The process used is entirely different to that requested for the week eight assignment.

The following example of code is a fully functioning implementation of the simple 'Caesar Cipher' method of encryption. The basis of the Caesar Cipher is described in the Week 8 material for the course. Whilst the encryption itself is fairly trivial and, as far as data encryption proper goes, incredibly weak; this implementation includes a number of interesting pieces of code to handle input and validation of input.

You do not need to copy out the code from this document in order to run it, a `.f90` version of the code can also be downloaded, ready to compile. Once downloaded to a Windows machine, it can be transferred using a program such as 'WinSCP', where you enter the name of the *Heron* server as with putty, and it permits you to transfer files between your current location and your *Heron* userspace.

The code, when compiled, can operate in a number of modes, determined automatically by the program based on the nature of its input. When run with no command line arguments, the program prompts the user to interactively enter both the text to be encrypted, and the key with which to encrypt it. If the code is supplied with a single command line argument, it first attempts to convert it into an integer. If it does so successfully, it assumes that you have supplied the key, not the text, assigns the command line argument as the key and prompts the user for text interactively. If it fails to convert the argument to an integer (the failure caught by the `IOSTAT` parameter), it assumes that it is the text which you have supplied, assigns it as such, and prompts the user to enter the key. The final possibility is that two command line arguments are entered. In this case, it applies the same process as for one argument, except in this case, whatever it determines the first argument to be (key or text) it assumes that the second argument is the other, assigns both key and text with that in mind, and does not prompt the user for any interactive input at all.

Additionally, when reading the supplied text, it checks to see if it starts with the code '`FILE=`'. If it does, it uses whatever follows the equals sign as a filename, and obtains the text to encode from that file.

So, if the program is run as follows, it will ask the user for the key and the text to be entered interactively:

```
:> ./cipher
```

If run as follows it will only ask for the text:

```
:> ./cipher 2
```

If run as follows it will only ask for the key:

```
:> ./cipher 'Some text'
```

If run as either of the following two instances it will not ask for anything, and will encrypt the supplied text string straight away:

```
:> ./cipher 2 'Some text'
:> ./cipher 'Some text' 2
```

And finally, if run as follows, it will not ask for anything, and will encrypt the contents of the file '`myfile.txt`' (if it exists):

```
:> ./cipher 2 FILE=myfile.txt
```

Have a look at the contents of the program code to see how it selects between these different modes of operation itself, determining the mode from the input already received.

The actual encryption is handled by simple character replacement from an array of the letters of the alphabet. Initially, the program converts all of the letters to lowercase, to avoid having to effectively check two different available alphabets later. Then, it works out the ciper based on the supplied key. This makes a new array with a shifted version of the alphabet. Finally, it goes through each letter in the original text supplied by the user. If the letter in in the regular alphabet (ie, not a symbol/space/numeral/etc) it finds that letter's position/index in the original, non-shifted, alphabet array. It then replaces that letter with whatever letter it finds at the same position/index in the ciper array.

This is not the fastest way to encrypt data, as it is required to search through many elements every time it needs to make a swap. The method asked for in the Week 8 assignment is more efficient. The method portrayed here was chosen because it is easier to initially understand (it is swapping in the same way a person would use a ciper, looking up a letter in a table and reading off what it should be swapped with). It was also chosen to avoid giving you a working version of the solution for the Week 8 assignment!

The content of the program itself begins on the next page. Remember that the `.f90` for this code is available to download separately, it is not necessary to copy the code from this document.

Contents of 'cipher.f90'

```
!------------------------------------------------------------------------------!
!-- Caesar cipher Encryption Program                                         --!
!-- Program Written by Tim Kinnear                                           --!
!-- This program performs a caesar cipher encryption procedure to a string --!
!-- of plain text supplied as either a command line argument, or input      --!
!-- interactively. The key for the cipher is also input either via the      --!
!-- command line or interactively.                                          --!
!------------------------------------------------------------------------------!

PROGRAM caesarcipher

  IMPLICIT NONE

  CHARACTER*512 :: plaintext, ciphertext, temptext
  CHARACTER*26 :: lowercase, uppercase, cipher
  CHARACTER*10 :: numerals
  CHARACTER*1 :: letter
  INTEGER :: plainlen, scanloc
  INTEGER :: key, testval, testio
  INTEGER :: nargs, i
  LOGICAL :: checkcontains

  !define character sets
  lowercase = 'abcdefghijklmnopqrstuvwxyz'
  uppercase = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
  numerals = '1234567890' !not actually used in current version, but retained for
                          !if their use in future is needed

  !get number of cmdline arguments
  nargs = IARGC()

  WRITE(*,*) '+---------------------------------------+'
  WRITE(*,*) "|   Tim's program of encryption magic   |"
  WRITE(*,*) '+---------------------------------------+'

  !if no cmdline args, ask for text and key interactively
  IF (nargs == 0) THEN
    CALL gettext(plaintext,plainlen)
    CALL getkey(key)
  !if one cmdline arg, test to see if integer, if it is, make it key, and ask for text,
  !otherwise, make it text and ask for key
  ELSE IF (nargs == 1) THEN
    CALL GETARG(1,temptext)
    READ(temptext,*,IOSTAT=testio) testval
    IF (testio == 0) THEN
      READ(temptext,*) key
      CALL gettext(plaintext,plainlen)
    ELSE
      plaintext = temptext
      plainlen = LEN(TRIM(plaintext))
      CALL getkey(key)
    END IF
  !if two cmdline args, test first to see if integer, if it is, make it key,
  !and use second for text, otherwise make first arg text, and test to see
  !if second is integer; if so, make second key, otherwise, quit
  ELSE IF (nargs == 2) THEN
    CALL GETARG(1,temptext)
    READ(temptext,*,IOSTAT=testio) testval
    IF (testio == 0) THEN
```

```
      READ(temptext,*) key
      CALL GETARG(2,plaintext)
      plainlen = LEN(TRIM(plaintext))
    ELSE
      plaintext = temptext
      plainlen = LEN(TRIM(plaintext))
      CALL GETARG(2,temptext)
      READ(temptext,*,IOSTAT=testio) key
      IF (testio .NE. 0) THEN
        WRITE(*,*) 'Could not determine which command line argument was key'
        STOP
      END IF
    END IF
  ELSE
    !if there are more than two cmdline arguments, tell user that is not valid
    WRITE(*,*) 'Maximum of two (2) arguments.'
    STOP
  END IF

  IF ((key < -26) .OR. (key > 26)) THEN
    WRITE(*,*) 'Key must be in one of the following ranges:'
    WRITE(*,*) 'For encryption, between 1 and 26'
    WRITE(*,*) 'For decryption, between -1 and -26'
    WRITE(*,*) 'To cycle through all possible keys, use 0 (zero)'
    STOP
  END IF

  !if the first 5 letters of the plaintext string are 'FILE=' then
  !assume that this precedes the name of a file that contains the
  !data to be encrypted, open that file and extract contents
  IF (plaintext(1:5) == 'FILE=') THEN
    OPEN(10,FILE=TRIM(plaintext(6:)))
    READ(10,'(A)') plaintext
    CLOSE(10)
    plainlen = LEN(TRIM(plaintext))
  END IF

  WRITE(*,*) '+---------------------------------------+'
  WRITE(*,*) "|      Plaintext and key prepared       |"
  WRITE(*,*) '+---------------------------------------+'

  !Echo choices to screen
  !when key is positive, 'encrypt' the data (shift forwards through cipher cycle)
  !when key is negative, 'decrypt' the data (shift backwards through cipher cycle)
  !when key is zero, go through all possible keys
  WRITE(*,'(A26,A,A)') 'Plaintext is: "', plaintext(1:plainlen), '"'
  WRITE(*,'(A26,I0)',ADVANCE='no') 'Key is:  ', key
  IF (key < 0) THEN
    WRITE(*,*) ' (decrypt)'
  ELSE IF (key > 0) THEN
    WRITE(*,*) ' (encrypt)'
  ELSE
    WRITE(*,*) ' (full slew)'
  END IF

  !change plaintext to uniform case
  CALL encrypt(plaintext,uppercase,lowercase,temptext)
  plaintext = temptext

  !if there are no regular lowercase characters in plaintext, then encryption/decryption
  !is not possible (caesar cipher only cycles through regular alphabet), tell user this
  !and exit
```

```fortran
  IF (.NOT. checkcontains(plaintext,lowercase)) THEN
    WRITE(*,*) 'Plaintext does not appear to contain any encryptable characters'
    STOP
  END IF

  !re inform user of the text being encrypted shifted to lowercase characters
  WRITE(*,'(A26,A,A)') 'Normalised case text is: "', plaintext(1:plainlen), '"'

  WRITE(*,*) '+---------------------------------------+'
  WRITE(*,*) "|        Performing Encryption          |"
  WRITE(*,*) '+---------------------------------------+'

  IF (key == 0) THEN
    !use all possible keys
    DO key = 1, 26, 1
      !create the cipher alphabet for specified key
      CALL gencipher(lowercase,key,cipher)
      !perform the encryption
      CALL encrypt(plaintext,lowercase,cipher,ciphertext)
      WRITE(*,'(A26,A,A,I0)') 'ciphered text is: "', TRIM(ciphertext), '" for key of ', key
    END DO
  ELSE
    !create the cipher alphabet for specified key
    CALL gencipher(lowercase,key,cipher)
    !perform the encryption
    CALL encrypt(plaintext,lowercase,cipher,ciphertext)
    WRITE(*,'(A26,A,A)') 'ciphered text is: "', TRIM(ciphertext), '"'
  END IF


  WRITE(*,*) '+---------------------------------------+'
  WRITE(*,*) "|         Encryption complete           |"
  WRITE(*,*) '+---------------------------------------+'

END PROGRAM caesarcipher

!Function which returns .TRUE. if 'string' contains any character specified by 'alphabet'
LOGICAL FUNCTION checkcontains(string,alphabet)
  IMPLICIT NONE
  CHARACTER*512, INTENT(IN) :: string
  CHARACTER*26, INTENT(IN) :: alphabet
  INTEGER :: i, scanloc
  !initialise check variable
  checkcontains = .FALSE.
  !go through each letter in the string supplied, and scan alphabet looking for that letter
  !if it returns non-zero, it means that that character from the string is in the alphabet
  !set var to .TRUE. to indicate success
  DO i = 1, LEN(TRIM(string)), 1
    scanloc = SCAN(alphabet,string(i:i))
    IF (scanloc .NE. 0) THEN
      checkcontains = .TRUE.
    END IF
  END DO
END FUNCTION checkcontains

!Main encryption routine (also decrypts, as effectively the same process)
SUBROUTINE encrypt(instring,alphabet,cipher,outstring)
  IMPLICIT NONE
  !original plaintext or encrypted text to be encrypted/decrypted
  CHARACTER*512, INTENT(IN) :: instring
  !text to send back to caller of the subroutine, with encrypted/decrypted version
  !of the original text
  CHARACTER*512, INTENT(OUT) :: outstring
  !alphabet is the original alphabet of characters in order,
  !cipher is the rearranged alphabet, shifted along by ‘key’ places
  CHARACTER*26, INTENT(IN) :: alphabet, cipher
  CHARACTER*1 :: letter
  INTEGER :: i, scanloc

  !initialise the string to return (fill with whitespace)
  outstring = REPEAT(' ',512)

  !go through each letter of original text, find the location of that letter in the alphabet string
  !once found, use the same location in the shifted cipher string to grab the enciphered replacement
  !for the original letter
  DO i = 1, LEN(TRIM(instring)), 1
    letter = instring(i:i)
    scanloc = SCAN(alphabet,letter)
    !if the letter in the plaintext is not in the alphabet, do not attempt to
    !replace it with anything
    IF (scanloc .NE. 0) THEN
      letter = cipher(scanloc:scanloc)
    END IF
    outstring(i:i) = letter
  END DO
END SUBROUTINE

!This subroutine generates the ciphered alphabet to perform encryption using
SUBROUTINE gencipher(alphabet,key,cipher)
  IMPLICIT NONE
  CHARACTER*26, INTENT(IN) :: alphabet !the alphabet which is going to be shifted
  CHARACTER*26, INTENT(OUT) :: cipher !the ciphered alphabet which is to be returned
  INTEGER, INTENT(IN) :: key !the key to apply the cipher
  INTEGER :: i, newi
  !initialise cipher, just in case
  cipher = REPEAT(' ',26)
  !go through each letter of the alphabet supplied
  DO i = 1, 26, 1
    !shift the index of the letter by ‘key’ (ie ‘1’ for ‘a’, with a key of 2 would become ‘3’ for ‘c’)
    newi = i + key
    IF (newi > 26) THEN
      !if the shift takes the new index past the end of the alphabet, subtract 26 to
      !loop back around from the beginning
      newi = newi - 26
    ELSE IF (newi < 1) THEN
      !if the shift takes the new index below the alphabet (for the case of decryption),
      !then add 26 to loop back to top
      newi = newi + 26
    END IF
    !the cipher character at the same index as being examined for the alphabet then becomes
    !the character at the shifted index of the alphabet
    cipher(i:i) = alphabet(newi:newi)
  END DO
END SUBROUTINE

!Subroutine to ask user for, and then read in, the text to be encrypted/decrypted
```

```
!no real error checking done here, as text could be practically anything
!could implement check for string being zero length (ie, nothing entered)
!however, this would be caught by various other checks later on (there will be nothing to encrypt!)
SUBROUTINE gettext(string, stringlen)
  IMPLICIT NONE
  CHARACTER*512, INTENT(OUT) :: string
  INTEGER, INTENT(OUT) :: stringlen
  WRITE(*,*) "Please enter plain text below, press return when done&
& (begin with code 'FILE=' to specify a filename to encrypt):"
  READ(*,'(A)') string
  stringlen = LEN(TRIM(string))
END SUBROUTINE

!Subroutine to ask user for, and then read in, the key for encryption/decryption
SUBROUTINE getkey(key)
  IMPLICIT NONE
  INTEGER, INTENT(OUT) :: key
  INTEGER :: io
  LOGICAL :: keyvalid = .FALSE.
  !keep looping until sensible input switched the keyvalid variable to .TRUE.
  DO WHILE (.NOT. keyvalid)
    WRITE(*,*) 'Please enter key number (1-26) below (negative numbers perform&
& decrypt for that key, 0 goes through all possible keys sequentially):'
    !Read in attempt
    READ(*,*,IOSTAT=io) key
    IF (io .NE. 0) THEN
      !if iostat returns error, key was not an integer (or nothing was entered/etc)
      !inform user, then let loop start again
      WRITE(*,*) 'Invalid key entered. (Non-integer)'
    ELSE IF ((key > -26) .AND. (key < 26)) THEN
      !if key is integer and in correct range, accept the value and switch keyvalid
      keyvalid = .TRUE.
    ELSE
      !case for key value being out of required range
      WRITE(*,*) 'Invalid key entered. (Invalid range)'
    END IF
  END DO
END SUBROUTINE
```

Example of program operation

```
:> ./cipher 5 'she sells sea shells on the sea shore'
```

```
 +-----------------------------------------+
 |   Tim's program of encryption magic    |
 +-----------------------------------------+
 +-----------------------------------------+
 |      Plaintext and key prepared        |
 +-----------------------------------------+
           Plaintext is: "she sells sea shells on the sea shore"
                 Key is:  5  (encrypt)
Normalised case text is: "she sells sea shells on the sea shore"
 +-----------------------------------------+
 |        Performing Encryption           |
 +-----------------------------------------+
        Cipered text is: "xmj xjqqx xjf xmjqqx ts ymj xjf xmtwj"
 +-----------------------------------------+
 |         Encryption complete            |
 +-----------------------------------------+
:> ./cipher 8 "Help me, Obi-Wan Kenobi, you’re my only hope"
```

```
 +---------------------------------------+
 |   Tim’s program of encryption magic   |
 +---------------------------------------+
 +---------------------------------------+
 |      Plaintext and key prepared       |
 +---------------------------------------+
           Plaintext is: "Help me, Obi-Wan Kenobi, you’re my only hope"
                 Key is:  8  (encrypt)
Normalised case text is: "help me, obi-wan kenobi, you’re my only hope"
 +---------------------------------------+
 |        Performing Encryption          |
 +---------------------------------------+
       ciphered text is: "pmtx um, wjq-eiv smvwjq, gwc’zm ug wvtg pwxm"
 +---------------------------------------+
 |         Encryption complete           |
 +---------------------------------------+
```

In this example, note how the non-alphabetical characters (the commas, hyphen and inverted comma) have not changed or moved position. Because this cipher is purely based on an alphabet shift; such character are not encrypted. The following example uses the output from the previous example, decrypting it with the correct key.

```
:> ./cipher -8 "pmtx um, wjq-eiv smvwjq, gwc’zm ug wvtg pwxm"
```

```
 +---------------------------------------+
 |   Tim’s program of encryption magic   |
 +---------------------------------------+
 +---------------------------------------+
 |      Plaintext and key prepared       |
 +---------------------------------------+
           Plaintext is: "pmtx um, wjq-eiv smvwjq, gwc’zm ug wvtg pwxm"
                 Key is:  -8  (decrypt)
Normalised case text is: "pmtx um, wjq-eiv smvwjq, gwc’zm ug wvtg pwxm"
 +---------------------------------------+
 |        Performing Encryption          |
 +---------------------------------------+
       ciphered text is: "help me, obi-wan kenobi, you’re my only hope"
 +---------------------------------------+
 |         Encryption complete           |
 +---------------------------------------+
```

This final example tries to decrypt the output from the second example without prior knowledge of the key, going through all of the possible keys sequentially. Note how the arguments for the program are the opposite way around to the previous example, first the text, then the key; this is not required for this mode, just an illustration of how the program is set up to determine for itself which way around the user has supplied the text and key.

```
:> ./cipher "pmtx um, wjq-eiv smvwjq, gwc’zm ug wvtg pwxm" 0
```

```
 +---------------------------------------+
 |   Tim’s program of encryption magic   |
 +---------------------------------------+
 +---------------------------------------+
 |      Plaintext and key prepared       |
 +---------------------------------------+
           Plaintext is: "pmtx um, wjq-eiv smvwjq, gwc’zm ug wvtg pwxm"
                 Key is:  0  (full slew)
Normalised case text is: "pmtx um, wjq-eiv smvwjq, gwc’zm ug wvtg pwxm"
 +---------------------------------------+
 |        Performing Encryption          |
 +---------------------------------------+
       ciphered text is: "qnuy vn, xkr-fjw tnwxkr, hxd’an vh xwuh qxyn" for key of 1
```

```
      ciphered text is: "rovz wo, yls-gkx uoxyls, iye'bo wi yxvi ryzo" for key of 2
      ciphered text is: "spwa xp, zmt-hly vpyzmt, jzf'cp xj zywj szap" for key of 3
      ciphered text is: "tqxb yq, anu-imz wqzanu, kag'dq yk azxk tabq" for key of 4
      ciphered text is: "uryc zr, bov-jna xrabov, lbh'er zl bayl ubcr" for key of 5
      ciphered text is: "vszd as, cpw-kob ysbcpw, mci'fs am cbzm vcds" for key of 6
      ciphered text is: "wtae bt, dqx-lpc ztcdqx, ndj'gt bn dcan wdet" for key of 7
      ciphered text is: "xubf cu, ery-mqd audery, oek'hu co edbo xefu" for key of 8
      ciphered text is: "yvcg dv, fsz-nre bvefsz, pfl'iv dp fecp yfgv" for key of 9
      ciphered text is: "zwdh ew, gta-osf cwfgta, qgm'jw eq gfdq zghw" for key of 10
      ciphered text is: "axei fx, hub-ptg dxghub, rhn'kx fr hger ahix" for key of 11
      ciphered text is: "byfj gy, ivc-quh eyhivc, sio'ly gs ihfs bijy" for key of 12
      ciphered text is: "czgk hz, jwd-rvi fzijwd, tjp'mz ht jigt cjkz" for key of 13
      ciphered text is: "dahl ia, kxe-swj gajkxe, ukq'na iu kjhu dkla" for key of 14
      ciphered text is: "ebim jb, lyf-txk hbklyf, vlr'ob jv lkiv elmb" for key of 15
      ciphered text is: "fcjn kc, mzg-uyl iclmzg, wms'pc kw mljw fmnc" for key of 16
      ciphered text is: "gdko ld, nah-vzm jdmnah, xnt'qd lx nmkx gnod" for key of 17
      ciphered text is: "help me, obi-wan kenobi, you're my only hope" for key of 18
      ciphered text is: "ifmq nf, pcj-xbo lfopcj, zpv'sf nz pomz ipqf" for key of 19
      ciphered text is: "jgnr og, qdk-ycp mgpqdk, aqw'tg oa qpna jqrg" for key of 20
      ciphered text is: "khos ph, rel-zdq nhqrel, brx'uh pb rqob krsh" for key of 21
      ciphered text is: "lipt qi, sfm-aer oirsfm, csy'vi qc srpc lsti" for key of 22
      ciphered text is: "mjqu rj, tgn-bfs pjstgn, dtz'wj rd tsqd mtuj" for key of 23
      ciphered text is: "nkrv sk, uho-cgt qktuho, eua'xk se utre nuvk" for key of 24
      ciphered text is: "olsw tl, vip-dhu rluvip, fvb'yl tf vusf ovwl" for key of 25
      ciphered text is: "pmtx um, wjq-eiv smvwjq, gwc'zm ug wvtg pwxm" for key of 26
+---------------------------------------+
|         Encryption complete           |
+---------------------------------------+
```

You can see how the real plaintext has been found for a key of 18. This is because it is working forwards through the alphabet. Because the cipher is cyclic, forward encryption of 18 is the same as backwards decryption of 8 ($26 - 8 = 18$).

Summary

Hopefully this code has shown you some of the various aspects of a more complicated code working in unison, rather than the smaller, isolated examples you have coded so far. The work from this point in the course onwards begins to become more complex and interlinked; assignments will not simply require implementation of the concept described in the Chapter, but to reuse concepts learnt from earlier in the course.

The week Eight material introduces 'strings' and 'formatting statements', both used extensively in this sample program. For more information on the Caesar Cipher used here, a description in more detail is given in the Week Eight material of this too. Week Eight combines the programming concepts of strings (stored text) along with several aspects of the simpler methods of encryption of information.