TEMASEK JUNIOR COLLEGE

Temasek Junior College

JC H2 Computing

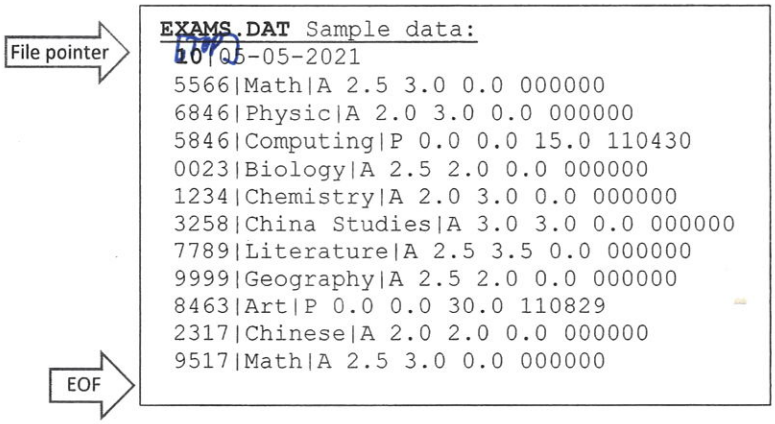
Problem Solving & Algorithm Design 9 – File Processing

1 File handling

1.1 Handling text files

Text files consist of lines of text that are read or written consecutively as strings.

It is good practice to explicitly open the files, stating the mode of operation, before reading from or writing to it. This is written as follows:



OPENFILE <File identifier> **FOR** <File mode>

The file identifier will usually be the name of the file. The following file modes are used:

- **READ**: for data to be read from the file
- **WRITE**: for data to be written to the file. A new file will be created and any existing data in the file will be lost.
- **APPEND**: for data to be added to the file, after any existing data.

A file should be opened in only **one mode** at a time.

Data is read from the file (after the file has been opened in READ mode) using the READFILE command as follows:

READFILE <File Identifier>, <Variable>

The Variable should be of data type STRING. When the command is executed, the next line of text in the file is read and assigned to the variable.

It is useful to think of the file as having a **pointer** which indicates the next line to be read. When the file is opened, the file pointer points to the first line of the file. After each READFILE command is executed the file pointer moves to the next line, or to the end of the file if there are no more lines.

The function **EOF** is used to test whether the file pointer is at the end of the file. It is called as follows:

EOF(<File Identifier>)

This function returns a Boolean value: TRUE if the file pointer is at the end of the file and FALSE otherwise.

Data is written into the file (after the file has been opened in WRITE or APPEND mode) using the WRITEFILE command as follows:

WRITEFILE <File identifier>, <String>

When the command is executed, the string is written into the file and the file pointer moves to the next line.

Files should be closed when they are no longer needed using the CLOSEFILE command as follows:

CLOSEFILE <File identifier>

1.2 Processing of sequential file.

Initial processing

OPENFILE <File identifier> **FOR** <File mode>

READFILE first record
DOWHILE records exist | **DOWHILE NOT EOF** | **DOWHILE** more data | **DOWHILE** more records
    Process this record
    READFILE next record
ENDDO

CLOSEFILE <File identifier>
Final processing

Example 1 This example uses the operations together, to copy all the lines from FileA.txt to FileB.txt, replacing any blank lines by a line of dashes.

```
DECLARE LineOfText : STRING

OPENFILE FileA.txt FOR READ
OPENFILE FileB.txt FOR WRITE

DOWHILE NOT EOF(FileA.txt) DO

     READFILE FileA.txt, LineOfText
     IF LineOfText = ""
          THEN
               WRITEFILE FileB.txt, "-----------------------"
          ELSE
               WRITEFILE FILEB.txt, LineOfText
     ENDIF

ENDDO

CLOSEFILE FileA.txt
CLOSEFILE FileB.txt
```

Example 2 Printing examination scores

A program is required to **read** and **print** a series of names and exam scores for students enrolled in a Computing course. The class average is to be **computed** and **printed** at the end of the report. Scores can arrange from 0 to 100. The last record contains a blank name and a score of 999, and is not to be included in the calculation.

A. Defining diagram

Input	Processing	Output
name	Read student details	name
exam_score	Print student details	exam_score
	Compute average score	average_score
	Print average_score	

	JOYCE TAN WAN LIN	65
	NG HUI TING JACQUELINE	45
	NUR RAMIZAH BTE RAMLI	77
	TANG KUAN YEE	67
	KAW TECK LIN	30
	KUNG GUANGJUN	90
	LOE CHUAN YUN	71
	OH JIEYI JOEL TIMOTHY	68
	PANG YINGXIANG BONNER	88
	TAY KAI ZHONG	56
		999

Data file: COMPUTING_SCORES

B. Solution algorithm

```
Set total_score to zero
Set total_students to zero

OPENFILE COMPUTING_SCORES.txt FOR READ
READFILE name, exam_score

DOWHILE exam_score not = 999
     Add 1 to total_students
     PRINT name, exam_score
     Add exam_score to total_score
     READFILE COMPUTING_SCORES.txt, name, exam_score
ENDDO

CLOSEFILE COMPUTING_SCORES.txt
IF total_students not = zero THEN
     average_score ← total_score / total_students
     PRINT average_score
ENDIF
```

C. Desk checking

(i) Input data

JOYCE TAN WAN LIN	65
NG HUI TING JACQUELINE	45
	999

(ii) Expected results

JOYCE TAN WAN LIN	65
NG HUI TING JACQUELINE	45
Average	

(iii) Desk check table:

Statement	total_score	total_students	name	exam_score	DOWHILE	average_score
Initialise	0	0				
READ			JOYCE TAN WAN LIN	65		
DOWHILE					true	
Add		1				
PRINT			yes	yes		
Add	65					
READ			NG HUI TING JACQUELINE	45		
DOWHILE					true	
Add		2				
PRINT			yes	yes		
Add	110					
READ			empty	999		
DOWHILE					false	
Compute						55
PRINT						yes

Example 3 Refer to mini Project in Modular Design

A module, **ADDSUBJECT,** which when called will allow the user to enter data on a new subject.

This data is then written to a text file named **EXAMS.DAT.**

EXAMS.DAT has the following structure:

```
<NoOfRecords><UpdateDate>
<SubjectCode><Name><SubType><P1Len><P2Len><PracLen><CDate>
<SubjectCode><Name><SubType><P1Len><P2Len><PracLen><CDate>
```

NoOfRecords is the number of records stored in file
UpdateDate is the date that the file was last updated and is in the form DD-MM-YYYY.

Discuss with details the **validations** need to apply for the data to be **captured** before they **store** into the file EXAM.DAT.

EXAMS.DAT Sample data:

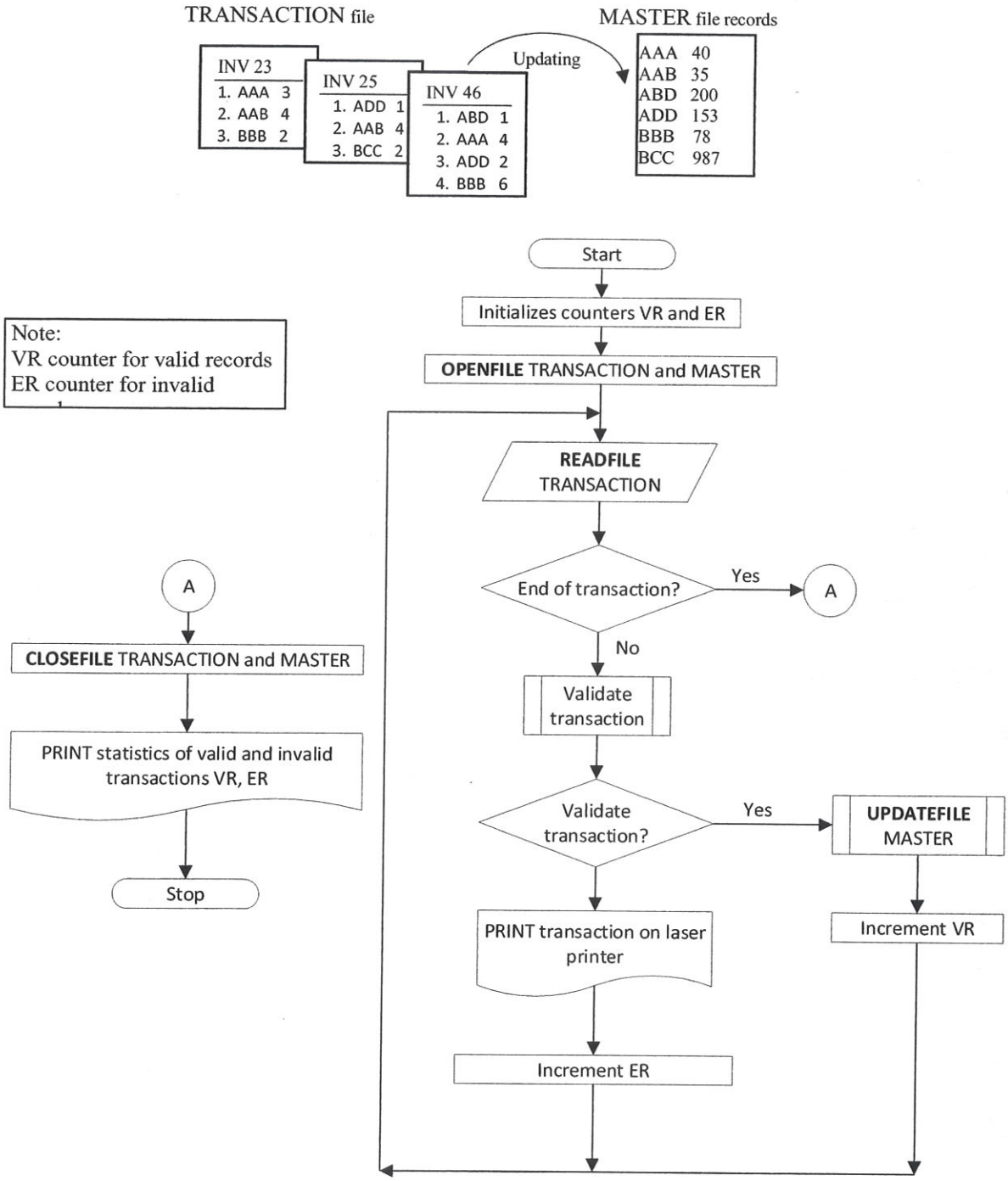
```
10|05-05-2021
5566|Math|A 2.5 3.0 0.0 000000
6846|Physic|A 2.0 3.0 0.0 000000
5846|Computing|P 0.0 0.0 15.0 110430
0023|Biology|A 2.5 2.0 0.0 000000
1234|Chemistry|A 2.0 3.0 0.0 000000
3258|China Studies|A 3.0 3.0 0.0 000000
7789|Literature|A 2.5 3.5 0.0 000000
9999|Geography|A 2.5 2.0 0.0 000000
```

```
PROCEDURE ADDSUBJECT()

   OutputFile = "EXAMS.DAT"
   OPENFILE OutputFile FOR WRITE

   NoOfRecords ← GetNoOfRecords()
   UpdateDate ← GetUpdateDate()

     WRITEFILE OutputFile, NoOfRecords, UpdateDate

  //LOOP NoOfRecords times
  DOWHILE NoOfRecords > 0

      CDate = "000000"
      SubjectCode ← GetSubjectCode()
      Name ← GetName()
      SubType ← GetSubType()

      //Handling different papers
      IF SubType = 'A' THEN          //Written papers
            PracLen ← 0.0
            P1Len ← GetP1Len()
            P2Len ← GetP2Len()
      ELSEIF SubType = 'P' THEN      //Practical subjects
            P1Len ← 0.0
            P2Len ← 0.0
            PracLen ← GetPracLen()
      ENDIF
      CDate ← GetCDate()
      NoOfRecords ← NoOfRecords - 1

      WRITEFILE OutputFile, SubjectCode, Name, SubType, P1Len1, P1Len2, PracLen, CDate

  ENDDO
  CLOSEFILE OutputFile

ENDPROCEDURE
```

Example 4

The following is the flowchart of a program module that reads a series of records from a transaction file and uses the valid transactions to update a master file.

Details of invalid transactions are output on a laser printer, together with a summary of the numbers of valid and invalid transactions processed.



1.3 Handling Random Files [self-reading]

Random files (also called binary files) contain a collection of data in their binary representation, normally as records of fixed length. They can be thought of as having a file pointer which can be moved to any location or address in the file. The record at that location can then be read or written.

Random files are opened using the RANDOM file mode as follows:

```
OPENFILE <File identifier> FOR RANDOM
```

As with text files, the file identifier will normally be the name of the file.

The SEEK command moves the file pointer to a given location:

```
SEEK <File identifier>, <address>
```

The address should be an expression that evaluates to an integer which indicates the location of a record to be read or written. This is usually the number of records from the beginning of the file. It is good practice to explain how the addresses are computed.

The command **GETRECORD** should be used to read the record at the file pointer:

```
GETRECORD <File identifier>, <Variable>
```

When this command is executed, the variable is assigned to the record that is read, and must be of the appropriate data type for that record (usually a custom type).

The command **PUTRECORD** is used to write a record into the file at the file pointer:

```
PUTRECORD <File identifier>, <Variable>
```

When this command is executed, the data in the variable is inserted into the record at the file pointer. Any data that was previously at this location will be replaced.

Example 5 The records from positions 10 to 20 of a file StudentFile.Dat are moved to the next position and a new record is inserted into position 10.
The example uses the custom type Student defined as.

```
TYPE Student
    DECLARE Surname : STRING
    DECLARE Name : STRING
    DECLARE DateOfBirth : STRING
    DECLARE CTGroup : STRING
    DECLARE GentreGroup : STRING
ENDTYPE

DECLARE Pupil : Student
DECLARE NewPupil : Student
DECLARE Position : INTEGER
NewPupil.Surname ← "Paik"
NewPupil.Name ←"Poh Leong"
NewPupil.DateOfBirth ←02/01/2000
NewPupil.CTGroup ←23
NewPupil.GentreGroup ←'M'

OPEN StudentFile.Dat FOR RANDOM
FOR Position = 20 TO 10 STEP -1
   SEEK StudentFile.Dat, Position
   GETRECORD StudentFile.Dat, Pupil
   SEEK StudentFile.Dat, Position + 1
   PUTRECORD StudentFile.Dat, Pupil
NEXT Position
SEEK StudentFile.Dat, 10
PUTRECORD StudentFile.Dat, NewPupil
CLOSE StudentFile.dat
```