

Cambridge IGCSE · Computer Science 0478 · Notes

File handling

Read online at pastopic.com/cambridge/computer-science-0478/notes/file-handling

Last checked 30 September 2026

Read first: [Programming concepts](#), [Procedures, functions and library routines](#)

© 2026 Pastopic. Free for personal study. Please share the link, not the file.

What you need to know

By the end of this topic you should be able to:

1. **Explain why a program keeps data in a file.** Variables disappear when a program stops; a file does not, so a program can save data in one and come back to it later.
2. **Write pseudocode that opens a file, reads from it or writes to it, and closes it.** The data can be a single value, such as a score or a surname, or a full sentence that fills one line of the file.

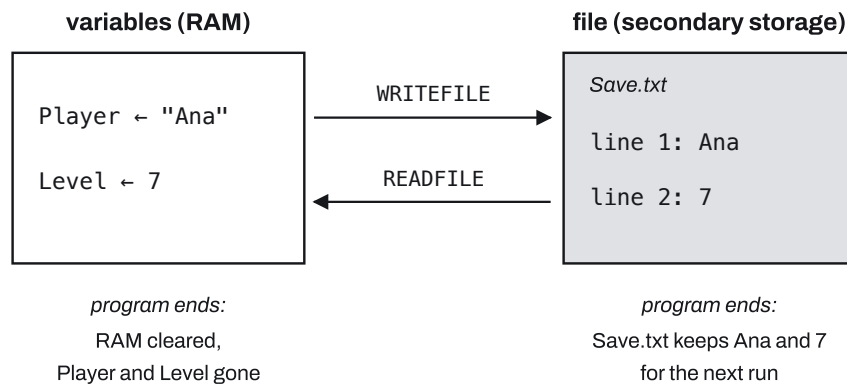
Declaring variables, data types and the string functions `LENGTH`, `UCASE` and `LCASE` are in the note on **Programming concepts**; writing procedures is in **Procedures, functions and library routines**. This note uses them where file questions need them.

Understand it

Where a program's data lives

While a program runs, every variable sits in **RAM**. RAM needs power to hold anything, and the operating system reuses a program's share of it as soon as the program finishes. So if a game keeps the player's name and level only in variables, both are lost the moment the game is closed, and the player has to start again next time.

A **file** is kept on **secondary storage**, such as a hard disk, an SSD or a memory stick, which holds its contents with the power off. A program can copy the values of its variables into a file before it stops, then copy them back into variables the next time it runs.



Storing data in a file is useful because:

- next time the program starts, it can load what it saved, so the user isn't asked for the same details twice; other software that knows the file's name can open it as well;
- files can be duplicated: put one on a memory stick or email it to use the data on a different machine, or keep a duplicate somewhere safe in case the first is damaged;
- secondary storage holds its contents with no power, so the data outlives the program run, which RAM cannot do.

Three steps: open, use, close

Every piece of file handling follows the same pattern.

1. **Open** the file, saying whether the program is going to take data out of it or put data into it. This is the file's **mode**.
2. **Use** it: read from it or write to it, as many times as needed.
3. **Close** it once the program has finished with it.

In Cambridge pseudocode the four keywords are **OPENFILE**, **READFILE**, **WRITEFILE** and **CLOSEFILE**:

```
OPENFILE <file name> FOR <mode>
READFILE <file name>, <variable>
WRITEFILE <file name>, <variable>
CLOSEFILE <file name>
```

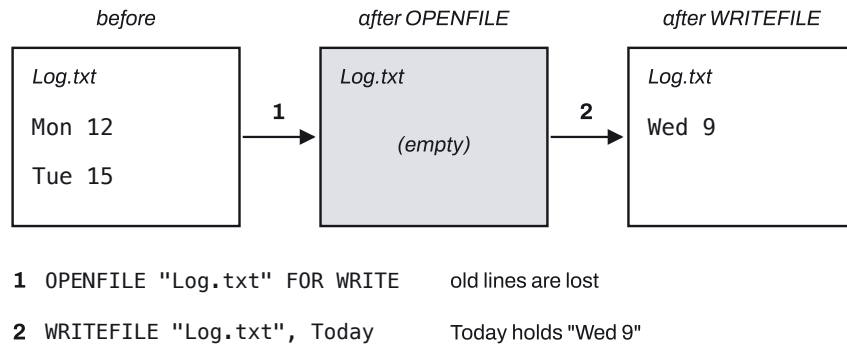
The mode is **READ** or **WRITE**. The file name is a string such as `"Save.txt"`; mark schemes also accept it written without the quotes, as in `OPENFILE Save.txt FOR READ`.

Writing to a file

```
DECLARE Planet : STRING
Planet ← "Neptune"
OPENFILE "Space.txt" FOR WRITE
WRITEFILE "Space.txt", Planet
CLOSEFILE "Space.txt"
```

After this runs, `Space.txt` holds one line: `Neptune`. The program can stop now and the word stays in the file.

Opening a file for `WRITE` starts it from empty. If `Space.txt` did not exist, it is created. If it did exist, whatever it held is thrown away the moment it is opened, before anything new is written. There is no mode in the 0478 pseudocode for adding to the end of an old file, so plan for this.



While a file stays open for `WRITE`, each `WRITEFILE` adds its item on a new line after the last one, so

```
DECLARE Colour : STRING
OPENFILE "Colours.txt" FOR WRITE
Colour ← "teal"
WRITEFILE "Colours.txt", Colour
Colour ← "amber"
WRITEFILE "Colours.txt", Colour
CLOSEFILE "Colours.txt"
```

leaves two lines in the file, `teal` then `amber`.

Reading from a file

```
DECLARE Greeting : STRING
OPENFILE "Welcome.txt" FOR READ
READFILE "Welcome.txt", Greeting
CLOSEFILE "Welcome.txt"
OUTPUT Greeting
```

`READFILE` takes one line out of the file and **stores it in the variable** named after the comma, just as an assignment would. The file itself doesn't change. If the first line of `Welcome.txt` is `Kia ora`, have a great day, then `Greeting` holds that whole line and it is output.

A line can be a single item, such as a number or one word, or a whole sentence; either way one `READFILE` takes one line. A second `READFILE` on the same open file takes the **next** line, so a program reads items back in the order they were written. Choose a variable of the right type: a whole number read into an `INTEGER`, text into a `STRING`.

Rules that catch people out

- **Open before use, close after.** Reading or writing a file that hasn't been opened is an error. Closing tells the computer the program has finished, so everything written is saved and the file is free for other programs.
- **Switching direction.** A file opened for READ can't be written to, and one opened for WRITE can't be read. To read a value and then save a new one in the same file, close the file and open it again with the other mode.
- **READFILE needs a variable** to put the data in; WRITEFILE needs the data to write, held in a variable.
- **The file name comes first** in READFILE and WRITEFILE, then a comma, then the variable.
- **INPUT and OUTPUT are not file commands.** INPUT takes data from the keyboard and OUTPUT shows it on the screen; files need READFILE and WRITEFILE.

Key facts and formulas

There is no formula sheet for Paper 2; learn these.

Fact	Status
OPENFILE <name> FOR READ or OPENFILE <name> FOR WRITE before a file is used	Learn it
READFILE <name>, <variable> copies the next line of the file into the variable	Learn it
WRITEFILE <name>, <variable> puts the variable's value into the file on a new line	Learn it
CLOSEFILE <name> when the program has finished with the file	Learn it
Opening for WRITE creates the file, or empties it if it already exists	Learn it
To switch between reading and writing, close the file and reopen it with the other mode	Learn it
Files keep data on secondary storage after the program ends; variables in RAM do not	Learn it
LENGTH (s) number of characters; UCASE (s) capitals; LCASE (s) small letters	Learn it

How to do it

In the Paper 2 questions we have tagged for this topic (8 questions, 2023–2025), 3 mix two types, so the counts add up to 11.

Question type	Questions
Write data to a file	3
Give reasons for storing data in a file	3
Recognise file-handling keywords and statements	3
Read data from a file	2

1. Write data to a file (3 questions)

Usually 3 marks for the file statements themselves, one for each, plus marks for whatever the question wraps around them (declaring variables, input, a procedure, string functions).

1. **Get the data into a variable first**, if the question asks for it: declare it, then `INPUT` it or work it out.
2. **Open the file for `WRITE`**, using the file name exactly as the question gives it.
3. **Write** with `WRITEFILE`, file name first, then the variable holding the data.
4. **Close** the file with `CLOSEFILE` and the same name.

For example, saving a temperature typed in by the user:

```
DECLARE Reading : REAL
INPUT Reading
OPENFILE "Temps.txt" FOR WRITE
WRITEFILE "Temps.txt", Reading
CLOSEFILE "Temps.txt"
```

Variations you will meet:

- **Inside a procedure with a parameter.** The parameter is the data to write, so there is no `INPUT`:

```
PROCEDURE SaveTeam(Team : STRING)
    OPENFILE "Teams.txt" FOR WRITE
    WRITEFILE "Teams.txt", Team
    CLOSEFILE "Teams.txt"
ENDPROCEDURE
```

Then use it with `CALL SaveTeam( "Falcons" )`, with the value in double quotes because it is a string. If part (a) asks you to declare variables, write one `DECLARE Name : TYPE` line each: whole number `INTEGER`, text `STRING`, true or false `BOOLEAN`.

- **Change the text before storing it.** Work out the new value in a variable, then write that variable: `Shout ← UCASE(Message)` then `WRITEFILE "Loud.txt", Shout`. `LENGTH(Message)` gives its number of characters if that is to be output too.
- **Two files.** Give each its own `OPENFILE`, `WRITEFILE` and `CLOSEFILE`, each with its own name, and write the right data into each one. Close **both**.
- **"Explain why the data needs to be stored in a file"** often comes straight after: see type 2.

2. Give reasons for storing data in a file (3 questions)

Usually 2 marks, one per reason. Pick two **different** ideas from this list and write each as a full sentence, tied to the scenario if there is one:

- **Nothing to re-enter.** On its next run the program loads what it saved, so the user skips typing the same details again; other software can open the file too.
- **Shareable and safe.** A file can be emailed or put on a memory stick for use on a different machine, and a duplicate held elsewhere protects against losing the original.
- **Not tied to RAM.** Variables vanish when the program stops or the power goes off; secondary storage holds the file's contents with no power.

For a password or a high score, for example: "the program can read it back from the file the next time it starts, instead of the user setting it again" and "a variable would lose it when the program closes, but secondary storage keeps it without power".

3. Recognise file-handling keywords and statements (3 questions)

Short items: a multiple-choice tick, matching descriptions to keywords, or picking out the file operations from a list of words.

- A file operation acts on a file: it opens it, takes data from it, puts data into it or closes it. A word for something done to data in memory, or with the keyboard or screen (such as *add*, *display*, *type* or *repeat*), is not a file operation.
- Match the job to the keyword: putting data **into** a file is `WRITE (WRITEFILE)`; getting data **out of** a file is `READ (READFILE)`; showing data on the screen is `OUTPUT`; typing data at the keyboard is `INPUT`.
- For a full statement, check that the keyword matches the direction the data moves, that the file name comes before the variable, and that a comma separates them. Only the four keywords exist; anything else that looks like one is not pseudocode. For example, `READFILE "Paint.txt", Colour` is right, but `READFILE Colour, "Paint.txt"` has the order swapped, `OPENFILE "Paint.txt", READ` is missing `FOR`, and `CLOSE "Paint.txt"` uses a keyword that doesn't exist.

4. Read data from a file (2 questions)

Usually 3–4 marks, one each for opening, reading, storing in the right variable and closing.

1. **Declare** a variable of a suitable type for what the file holds, if asked.
2. **Open the file for READ.**
3. **Read** with `READFILE`, file name first, then the variable the question names.
4. **Close** the file.
5. **Then use the variable** as the question asks, such as outputting it, changing it with `UCASE` or `LCASE`, or finding its `LENGTH`.

For example, a kitchen timer loads its last setting, a whole number of seconds, and shows it in minutes and seconds:

```
DECLARE TimerSeconds : INTEGER
OPENFILE "Timer.txt" FOR READ
READFILE "Timer.txt", TimerSeconds
```

```
CLOSEFILE "Timer.txt"
OUTPUT DIV(TimerSeconds, 60), " min ", MOD(TimerSeconds, 60), " s"
```

If the file held 135, this outputs 2 min 15 s, since $135 = 2 \times 60 + 15$.

Variations you will meet:

- **Several items:** one READFILE for each line, in the order they are in the file, each into its own variable.
- **Read, change, save back:** open for READ, read, **close**; change the value; then open for WRITE, write, close.

Worked examples

The bold codes show where marks are earned, as in Cambridge mark schemes. The number is how many marks.

- **B1** mark for a correct result on its own, with no method needed.
-

Example 1

A fitness app keeps the number of steps a user has walked today in the integer variable `Steps`.

(a) Write pseudocode to store the value of `Steps` in a text file called `StepLog.txt`. **[3]**

(b) `StepLog.txt` held 10032. The code from (a) is then run with `Steps = 8214`. State what `StepLog.txt` holds afterwards, and explain why. **[2]**

(a)

```
OPENFILE "StepLog.txt" FOR WRITE
WRITEFILE "StepLog.txt", Steps
CLOSEFILE "StepLog.txt"
```

- The file opened with its name and FOR WRITE. **B1**
- WRITEFILE with the file name then Steps. **B1**
- The file closed. **B1**

(b) Only 8214 **B1**. Opening the file for WRITE empties it first, so 10032 is removed before 8214 is written. **B1**

Example 2

A museum's information screen reads its settings from `Kiosk.txt`. The first line holds the screen's language, such as FR. The second line holds how long the screen waits before it goes blank, in whole seconds.

Write pseudocode that declares suitable variables, reads both settings from the file, and outputs the language and the waiting time in minutes, rounded to 1 decimal place. **[6]**

```

DECLARE Language : STRING
DECLARE WaitSeconds : INTEGER
OPENFILE "Kiosk.txt" FOR READ
READFILE "Kiosk.txt", Language
READFILE "Kiosk.txt", WaitSeconds
CLOSEFILE "Kiosk.txt"
OUTPUT "Language: ", Language
OUTPUT "Blank after ", ROUND(WaitSeconds / 60, 1), " minutes"

```

- Both variables declared with suitable types (text and a whole number). **B1**
- The file opened FOR READ. **B1**
- The language read into its variable with READFILE. **B1**
- A second READFILE for the waiting time, in the order of the lines in the file. **B1**
- The file closed. **B1**
- Both outputs, with the seconds divided by 60 and rounded to 1 place. **B1**

If the file holds FR and then 150, the screen shows Language: FR and, since $150 \div 60 = 2.5$, Blank after 2.5 minutes.

Example 3

```

DECLARE Pet : STRING
DECLARE First : STRING
OPENFILE "Pets.txt" FOR WRITE
Pet ← "rabbit"
WRITEFILE "Pets.txt", Pet
Pet ← "gecko"
WRITEFILE "Pets.txt", Pet
CLOSEFILE "Pets.txt"
OPENFILE "Pets.txt" FOR READ
READFILE "Pets.txt", First
CLOSEFILE "Pets.txt"
Pet ← "parrot"
OPENFILE "Pets.txt" FOR WRITE
WRITEFILE "Pets.txt", Pet
CLOSEFILE "Pets.txt"
OUTPUT First

```

(a) State what is output. **[1]**

(b) State what Pets.txt holds when the program ends. **[1]**

(c) Explain why the file has to be closed before it is opened for READ. [2]

(a) rabbit **B1**: the one READFILE takes the first line.

(b) Just parrot **B1**. The last OPENFILE ... FOR WRITE wiped rabbit and gecko.

(c) While Pets.txt is open FOR WRITE, it can only be written to, so a READFILE on it would be an error **B1**. Closing it and then reopening it FOR READ switches it to reading **B1**.

Example 4

A quiz keeps the highest score ever reached, a whole number, in the file Record.txt.

Write pseudocode that:

- reads the highest score from Record.txt
- inputs the score just achieved
- if the new score is higher, stores it in Record.txt in place of the old one and outputs New record
- otherwise outputs the score still to beat.

Declare the variables you use and close the file each time you have used it. [7]

```

DECLARE Record : INTEGER
DECLARE Score : INTEGER
OPENFILE "Record.txt" FOR READ
READFILE "Record.txt", Record
CLOSEFILE "Record.txt"
INPUT Score
IF Score > Record
    THEN
        OPENFILE "Record.txt" FOR WRITE
        WRITEFILE "Record.txt", Score
        CLOSEFILE "Record.txt"
        OUTPUT "New record"
    ELSE
        OUTPUT "Score to beat: ", Record
ENDIF

```

- Both variables declared as INTEGER. **B1**
- Record.txt opened for READ and closed after reading. **B1**
- The old record read into a variable with READFILE. **B1**
- The new score input and compared with the record. **B1**
- The file opened again, this time for WRITE, only when the score is higher. **B1**
- The new score written with WRITEFILE and the file closed. **B1**
- Both messages output in the right branches. **B1**

With 17 in the file: a score of 21 replaces it and outputs `New record`; a score of 12 leaves 17 in the file and outputs `Score to beat: 17`. Opening for `WRITE` only inside the `IF` matters: opening it before the test would empty the file even when the record stands.

Common mistakes

- **Leaving out the mode.** `OPENFILE "Data.txt"` on its own loses the mark; it must end `FOR READ` or `FOR WRITE`.
- **Opening in the wrong mode**, such as `FOR READ` and then `WRITEFILE`. Decide first whether data goes in or comes out.
- **Swapping the order** after `READFILE` or `WRITEFILE`. The file name comes first and the variable second.
- **Using `INPUT` or `OUTPUT` for files.** `INPUT` reads the keyboard and `OUTPUT` writes to the screen, so neither earns a mark where a file statement is needed.
- **Forgetting to close**, or closing only one of two files. When the question mentions closing, it is a separate mark.
- **Reading without storing.** `READFILE "Scores.txt"` with no variable throws the data away; name the variable the question gives.
- **Expecting `WRITE` to add to the end.** It empties the file first. Reading and then writing back needs a close and a reopen.
- **Changing the given names.** If the question says `Attend.txt` and `Visitor`, use exactly those.
- **Vague reasons.** "So it is saved" earns little. Say what the file gives that a variable can't: the data stays after the program ends and can be read back later.

How to get the marks

- **File code is marked one statement at a time:** open with the right mode, read or write with the right variable, close. Write each on its own line so the examiner can tick each one.
- **Declarations are a separate mark** when the question says "declare": one `DECLARE` per variable, with a suitable type.
- **Use the question's names exactly**, for files, variables and procedures.
- **"Write pseudocode"** means pseudocode, not a programming language: `READFILE`, not `open ( )` or `readline ( )`.
- **"Give two reasons"** needs two different ideas, not the same idea said twice ("it is kept" and "it isn't lost" are one idea).
- **"Explain"** needs a reason with its effect: "the data is on secondary storage, so it is still there after the program ends".
- **Minor slips are ignored**, such as missing quotes round a file name, but a missing keyword or mode is not.

Quick check

1. Write the statement that opens `Menu.txt` so a program can take data from it, and the statement that then stores its first line in `Dish`.
2. This code should store in `Towns.txt` a town typed in by the user. It has three errors. Identify and correct them.

```
DECLARE Town : STRING
INPUT Town
OPENFILE "Towns.txt"
READFILE "Towns.txt", Town
```

3. A to-do list app saves the list of tasks in a file instead of keeping them only in variables. Give two reasons why.
4. `Code.txt` holds three lines: `A7`, `B2`, `C9`. State what this outputs.

```
DECLARE P : STRING
DECLARE Q : STRING
OPENFILE "Code.txt" FOR READ
READFILE "Code.txt", P
READFILE "Code.txt", Q
CLOSEFILE "Code.txt"
OUTPUT Q, P
```

5. `Visits.txt` holds a whole number: the number of times a web page has been visited. Write pseudocode that adds 1 to the number in the file.

Answers

1. `OPENFILE "Menu.txt" FOR READ`, then `READFILE "Menu.txt", Dish`.
2. The `OPENFILE` has no mode: `OPENFILE "Towns.txt" FOR WRITE`. `READFILE` takes data out, but the town has to go in: `WRITEFILE "Towns.txt", Town`. The file is never closed: add `CLOSEFILE "Towns.txt"` at the end.
3. Any two: when the app opens again it loads the saved tasks, so the user doesn't retype the whole list; the list can be synced or copied to the user's other devices, and a duplicate guards against losing it; tasks held only in variables disappear when the app closes, but the file sits on secondary storage, which needs no power.
4. B2A7: P gets the first line, A7, and Q the second, B2; C9 is never read.
5. Read, close, change, then reopen for writing:

```

DECLARE Visits : INTEGER
OPENFILE "Visits.txt" FOR READ
READFILE "Visits.txt", Visits
CLOSEFILE "Visits.txt"
Visits ← Visits + 1
OPENFILE "Visits.txt" FOR WRITE
WRITEFILE "Visits.txt", Visits
CLOSEFILE "Visits.txt"

```

If the file held 41, it now holds 42.

Past questions to try

- **0478/21/O/N/24 Q8**: write an accepted password to a file, then explain why it has to be stored in one.
- **0478/23/O/N/23 Q3**: match descriptions to keywords such as `WRITE`, `READ` and `OUTPUT`, and give reasons for using a file.
- **0478/21/M/J/24 Q7**: why files are useful, then read a text file and output it in capitals with its length.