

Cambridge AS & A Level · Computer Science 9618 · Notes

System software AS

Read online at pastopic.com/cambridge/computer-science-9618/notes/system-software-as

Last checked 30 September 2026

Read first: [Hardware](#), [Processor fundamentals](#)

© 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 computer needs an Operating System (OS).**
2. **Explain the OS's key management tasks:** memory management, file management, security management, hardware management (input, output and peripherals) and process management.
3. **Explain why an OS comes with utility software**, and what these utilities do: disk formatter, virus checker, defragmentation software, disk contents analysis and disk repair software, file compression and back-up software.
4. **Explain program libraries:** new software is often built partly from existing code in libraries, and using library files, including Dynamic Link Library (DLL) files, benefits the developer.
5. **Explain why each translator is needed:** an assembler for assembly language, a compiler for a high-level language, and an interpreter that translates and runs a high-level language program.
6. **Give the benefits and drawbacks of a compiler and of an interpreter**, and justify using each one.
7. **Know that some high-level languages are partly compiled and partly interpreted**, such as Java (console mode).
8. **Describe the features of a typical Integrated Development Environment (IDE):** for coding (context-sensitive prompts), for first error detection (dynamic syntax checks), for presentation (prettyprint, expanding and collapsing code blocks) and for debugging (single stepping, breakpoints, watching variables and expressions, the report window).

How the OS pages memory, schedules processes in detail and how a compiler works stage by stage come later, in the A Level topic on system software.

Understand it

Why a computer needs an Operating System

System software runs the computer itself; **application software** (a word processor, a game) does a job for the user. The **Operating System** is the most important piece of system software. Without one, every program would have to know exactly how to drive the disk, the screen and the keyboard of that one machine, and two

programs could not run together safely.

The OS sits between the user, the applications and the hardware. It:

- **provides a user interface** (a graphical interface or a command line), so the user can communicate with the computer;
- **hides the complexity of the hardware**: a program asks the OS to "save this file" and the OS deals with the disk;
- **provides a platform** that application software runs on;
- **manages the computer's resources**: processor time, memory, storage and devices, shared fairly between programs and users.

The last point splits into five **key management tasks**. Learn the five names exactly as the syllabus gives them: exams often name one and ask for others.

The five key management tasks

Memory management looks after **main memory (RAM)**, not the disk.

- It **allocates** a part of RAM to each program (process) when it starts, and **frees** that part when it ends.
- It keeps track of which parts of RAM are in use and which are free.
- It keeps the data of all running programs in RAM **at the same time** without one **overwriting** another, and stops one process reading or changing another's memory.
- It decides which processes, and which parts of them, are in main memory at any moment, using memory efficiently.

File management looks after files on **secondary storage**.

- It sets up the **file system** (for example FAT or NTFS): storage is divided into **allocation units** (blocks), and the OS records which blocks belong to which file and which are free.
- It creates and maintains the **directory (folder) structure** and the **file naming conventions**.
- It provides the operations on files: create, open, save, close, copy, move, rename, delete, and **search** for a file.
- It controls **access** to files, by users and by other software, with access rights, so files can be shared safely.

Security management protects the system and its data.

- It **authenticates** users and processes, for example with user accounts and passwords, so there is no unauthorised access.
- It applies **access rights** (who may read, change or delete what).
- It makes provision for **recovering** lost data, installs **security updates** to the OS, and keeps **logs** of activity so problems can be traced.

Hardware management (input, output and peripherals) lets the computer use its devices.

- It installs and uses **device drivers**: a driver is the software that translates the OS's general instructions into the commands that one particular device understands.
- It **sends data to** output devices and **receives data from** input devices.
- It manages **buffers**, so a fast processor and a slow device (a printer) can exchange data smoothly, and **queues** such as the print queue.
- It handles **interrupts** from devices, such as "out of paper" or "key pressed".

Process management looks after the programs that are running. A **process** is a program while it is being executed; a word processor, a music player and the OS's own tasks are all processes.

- It **schedules** processes: it decides which process runs on the processor next and for how long.
- It makes **multitasking** possible: the processor switches between processes so quickly that they seem to run at the same time. One process is paused while another uses the processor.
- It gives each process **fair access** to the processor, taking **priorities** into account, and handles **interrupts**.
- It **allocates the resources** each process needs (memory, devices, files).
- It lets processes **share data** with each other and **prevents interference** between them, resolving conflicts when two want the same resource.

Memory management and process management work together for multitasking: memory management keeps every running program in RAM at once, each in its own protected area, and process management takes turns giving them the processor.

Utility software

Utility software helps the user **set up, maintain, analyse and optimise** the computer, so it keeps running correctly and quickly. Utilities are part of the system software, usually supplied with the OS. Know what each one does **and why it helps**.

Utility	What it does	Why it is needed
Disk formatter	Prepares a disk for first use: sets up the file system and its allocation table, can split the disk into partitions, checks for faulty areas; formatting also removes existing data	A new disk cannot store files until it has a file system
Virus checker	Scans files and memory for malicious code, compares them with known malware, then quarantines or deletes it; must be updated often	Malware can corrupt or delete data, or let others in
Defragmentation software	Moves the scattered pieces of each file on a hard disk so that each file is stored in contiguous (neighbouring) blocks, and gathers the free space together	Faster file access, so better performance

Utility	What it does	Why it is needed
Disk contents analysis / disk repair	Checks the disk for errors, marks bad sectors so they are not used again, fixes inconsistencies in the file system and recovers data where it can; the analysis shows how space is used (large, duplicate or unused files)	Restores corrupted files and lost data; frees space
File compression	Makes files smaller	Less storage used; quicker to send, using less bandwidth
Back-up software	Copies files to another device or place, often automatically on a schedule	Lost or corrupted data can be restored , and the user doesn't have to remember

How defragmentation helps is asked often. When files are saved, grow and are deleted over time, a new file has to be fitted into whatever gaps are free, so it ends up in **fragments** scattered across the disk. To read a fragmented file, the read/write head has to move to each fragment in turn.

Before: file A is in 4 fragments



defragmentation moves the blocks

After: file A is in 1 contiguous run



A, B, C: blocks of three files · dashed: free block

After defragmentation each file is in one run of blocks: reading file A in the diagram needs 1 head movement instead of 4. So the head moves less, each file is read faster, and the computer performs better.

Defragmentation is for **magnetic hard disks**: a solid state drive has no moving head, so it gains almost nothing, and the extra writing wears it out.

How a compression method works (run-length encoding, for example) belongs to Information representation; here you only need why compression is useful.

Program libraries and DLLs

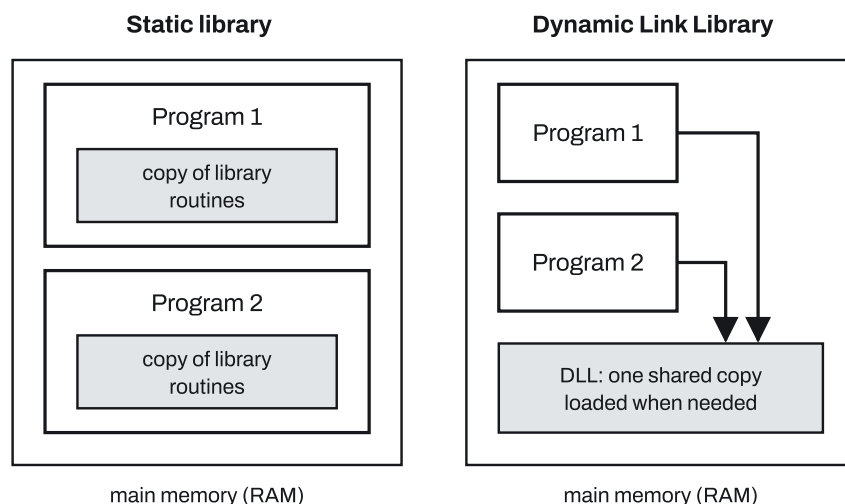
Programmers rarely write everything from scratch. A **program library** is a collection of **pre-written, pre-compiled and pre-tested subroutines** (procedures and functions) that can be **imported** into, or **called** from, other programs. A program that needs a square root, a sorting routine or a way to draw a window simply calls the library routine.

What the programmer gains:

- **less code to write**, so the program is finished sooner;
- **less testing**, because the routines are already tested, so they are **more likely to work**;
- routines they **could not write themselves** (complex maths, graphics, encryption);
- **more readable** code, because a call replaces many lines;
- in a team, routines written once can be **shared and reused** by everyone.

The drawbacks: a routine may not do **exactly** what is needed (and then has to be edited and retested), it may have **compatibility** problems with the program, it might contain **bugs** the programmer cannot see, and if someone else later **changes** it, the program may start giving unexpected results.

A library can be joined to a program in two ways. With an ordinary (static) library, the routines the program uses are **copied into the executable file** when it is built. A **Dynamic Link Library (DLL)** file stays **separate**: the program links to it only when it runs, and the DLL is loaded into memory **once, when it is needed**.



So a DLL gives particular benefits:

- the **executable file is smaller**, because it doesn't contain the library routines;
- **main memory is saved**: one copy of the DLL in RAM is shared by every program that uses it;
- the DLL can be **updated or corrected on its own**, and every program that uses it gets the change **without being recompiled**;
- the programmer **doesn't maintain** the DLL code, because it is separate from the program.

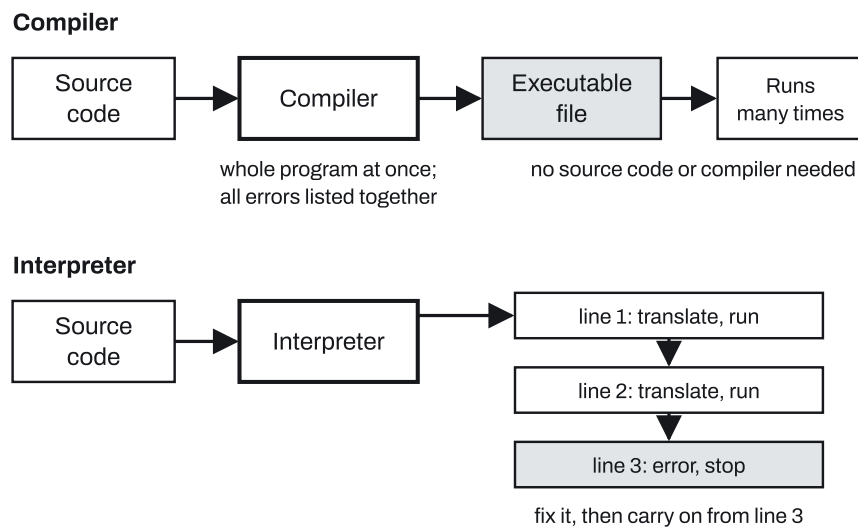
The catch is that the program depends on the DLL: if the DLL is missing, or is replaced by a version that doesn't match, the program fails.

Language translators

The processor only runs **machine code** (binary instructions). Every other program has to be **translated** first.

- An **assembler** translates **assembly language** (mnemonics such as LDD and ADD) into machine code. Each assembly instruction usually becomes one machine code instruction.

- A **compiler** translates a whole **high-level language** program (the **source code**) into machine code (**object code**) in one go, before it runs.
- An **interpreter** translates **and runs** a high-level language program **one statement at a time**.



How a compiler works. It checks and translates **all** the source code first. If it finds errors, it lists them **all together** at the end, and nothing runs. If there are none, it produces an **executable file**, which runs **without the source code and without the compiler**, and runs fast because translation is already done.

How an interpreter works. It translates one line and **runs it straight away**, then moves to the next line. At a line with an error it **stops** and reports that one error. The programmer can correct it **immediately** and the interpreter carries on from that point. No executable file is made, so the program has to be translated again **every time it runs**, and the user needs the interpreter and the source code.

	Compiler	Interpreter
Translates	The whole program before it runs	One statement at a time, running each
Errors	All reported together after translation; one error can trigger a list of false follow-on errors	Stops at the first error; fixed in real time, then continues from there
Output	An executable file	Nothing kept
To run later	No translator or source code needed	Interpreter and source code needed every time
Speed of running	Fast: translated once	Slower: translated every time
Unfinished code	Cannot be translated until it has no errors	Parts can be tested before the rest is written
Source code	Stays with the programmer	Has to be given out with the program

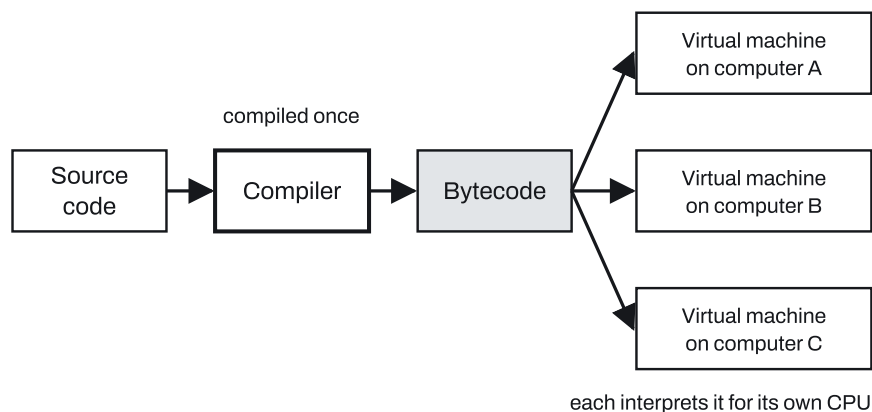
Which to use when. The two translators suit different stages, and a programmer often uses both.

- **While writing and debugging, use an interpreter.** It stops at an error so it can be fixed straight away and the program continued from there; errors come one at a time, so there are no knock-on errors; the effect of each change is seen immediately; and part of a program can be tested before the rest is written.
- **When the program is finished, use a compiler.** It makes an executable file that can be run and tested many times without translating again; customers can run it without the translator; and the source code isn't handed out, so it can't easily be copied, changed or sold by others.

Compiling a large program takes time, and it must be recompiled after every change, so a compiler is slow to use while code is still changing.

Partly compiled, partly interpreted

Some languages, such as Java (console mode), use **both**. The source code is **compiled into intermediate code** (Java calls it **bytecode**), which is not the machine code of any real processor. On each computer, a **virtual machine** then **interprets** the bytecode when the program runs, turning it into machine code for that computer's processor.



Why do it this way?

- **Portability:** the same intermediate code runs on **any platform** that has the virtual machine, so the source code doesn't need recompiling for each kind of computer.
- Translating intermediate code at run time is **quicker than interpreting source code**, because the main checking and translating are already done.
- The machine code is made at run time, so it can be **optimised for the processor** actually running it.

Don't confuse this with "use an interpreter while developing, then a compiler at the end". That is a different idea (see above), and it scores nothing when the question is about partial compilation.

Integrated Development Environments (IDEs)

An **IDE** is one program that holds everything needed to write, translate, run and test code: an editor, translator(s) and debugging tools. Its features are grouped by what they help with. Learn each one's **name** and **what it does for the programmer**.

For coding:

- **Context-sensitive prompts:** as you type, the IDE suggests what could come next (keywords, the names of your variables and subroutines, the parameters a subroutine needs), so you can pick one instead of remembering or typing it all. **Auto-completion** and **auto-correct** of simple mistakes do a similar job.

For initial error detection:

- **Dynamic syntax checks:** each line is checked **as it is typed**, and statements that break the language's rules are **underlined or highlighted** at once, before the program is ever run.

For presentation (making code easy to read):

- **Prettyprint:** keywords, variables, strings and comments are shown in different **colours**, with consistent indentation, so the structure is clear and mistakes stand out.
- **Expand and collapse code blocks:** a block (a subroutine, a loop) can be hidden to one line and opened again, so you see the overall structure and only the part you are working on.
- **Automatic indentation (formatting):** code is indented as you type, showing which statements are inside which block.

For debugging (finding logic and run-time errors):

- **Single stepping:** runs the program **one line (statement) at a time**, so you can see the effect of each line and follow the flow of the program.
- **Breakpoints:** the program runs normally until a **chosen line**, then **pauses**, so you can inspect the values at that point.
- **Variable and expression watch window:** shows the current values of chosen variables and expressions and how they **change** as the program runs.
- **Report window:** shows the program's output and the contents of variables and data structures while it runs.

The first three groups help while **writing** the code; the debugging tools help while **testing** it.

Key facts and formulas

There are no formulas in this topic, and the exam gives no formula list for it, so every fact below is **Learn it**.

Fact	Status
Five key management tasks: memory, file, security, hardware (input/output/peripherals), process	Learn it
Six utilities: disk formatter, virus checker, defragmentation, disk contents analysis / disk repair, file compression, back-up	Learn it
Three translators: assembler, compiler, interpreter	Learn it

Fact	Status
IDE features: coding (context-sensitive prompts), error detection (dynamic syntax checks), presentation (prettyprint, expand/collapse), debugging (single stepping, breakpoints, watch and report windows)	Learn it

How to do it

In the Paper 1 questions we have tagged for this topic (32 questions, 2021–2025), the types came up this often. 22 of the 32 questions mix more than one type, so the counts add up to more than 32.

Question type	Questions
Name or describe IDE features	12
Name, match or explain utility software	11
Program libraries and DLLs	10
Compare a compiler and an interpreter, or justify which to use	9
Name the OS key management tasks or state what an OS is for	8
Describe what one OS management task does	8
Describe how a compiler, interpreter or assembler works	4
Explain why a language is partly compiled and partly interpreted	2

The facts are all in "Understand it"; what earns the marks is **applying them to what the question asks**.

1. Name or describe IDE features (12 questions)

1. **Find the category** the question wants: coding, presentation or debugging (or "writing" versus "testing"). A correct feature from the wrong category scores nothing.
2. **Name the feature** with its proper name, such as "breakpoint" or "prettyprint".
3. **Describe what it does for the programmer**, not what its name says. "A breakpoint is where the program breaks" repeats the name; "stops the program at a chosen line so the variable values can be checked" earns the mark.

Variations you will meet:

- **A table with one feature per purpose**, "each feature must be different", and sometimes "not a translator". Use context-sensitive prompts (coding), prettyprint or expand/collapse (presentation), single stepping or breakpoints (debugging).
- **"Other than ..."**: if prettyprint or single stepping is already given, pick another from that group (expand/collapse, auto-indent; breakpoints, watch window).

- **Describe named features** (a four-row table) or **match features to descriptions**. Keep context-sensitive prompts (suggest code) apart from dynamic syntax checks (highlight errors): examiners see these swapped.
- **"How can the programmers use the debugging features?"** Describe using them: set a breakpoint before the suspect line, run to it, then single-step while watching the variables change.

2. Name, match or explain utility software (11 questions)

1. **Name the utility** exactly (the six in the table in "Understand it").
2. **Say what it does**, then **why that helps**, as a chain. For "how does it improve performance", the last link must be about performance, for example "... so less head movement, so files are read faster".

Variations you will meet:

- **Matching lines** from utilities to their purposes (there's a spare description). Defragmentation is "makes files contiguous", **not** "makes files smaller" (that is compression); a formatter "prepares or initialises a disk".
- **Pick a utility for a situation**: corrupted or lost files → disk repair, back-up; a slow hard disk → defragmentation; a new disk → formatter; files too big to send → compression.
- **A utility that is not about security**, or **two that improve performance**: defragmentation, compression, disk contents analysis (frees space); give the reason for each.
- **Explain why back-up software is needed**: data can be restored when lost; copies are made regularly and automatically.
- **"Describe the purpose of utility software"** in general: it helps set up, configure, maintain and optimise the computer, for example by checking for faults or making storage use more efficient. A list of utilities doesn't answer this.
- **Benefits of compressing a file before sending it**: less storage, faster transfer, less bandwidth. How a compression method works belongs to Information representation.

3. Program libraries and DLLs (10 questions)

1. **Define** a library with the three "pre-" words: pre-written (and pre-compiled, pre-tested) subroutines that can be imported or called from other programs. "Programs anyone can use" is too vague.
2. **Benefits and drawbacks**: take them from "Understand it", and write each as a point then its "so" (for example "already tested, **so** less testing time"), because many mark schemes pair a benefit with its expansion.

Variations you will meet:

- **"Benefits of a DLL"** (or "of a library with DLL files instead of one without"). Give **DLL-only** points: smaller executable; less **memory**, because one copy is loaded when needed and shared; the DLL can be updated without **recompiling** the program; no maintenance by the programmer. Examiners reject general library benefits here, and "saves space" or "faster" without saying which space (memory or disk) and why.

- **"How can a library be used while writing a program?"**: describe the use (import the library, call its routines instead of writing them), not just the benefits.
- **Benefits to a team** that creates its own library: routines shared and reused by all without rewriting or retesting, and everyone uses the others' specialisms.

4. Compare a compiler and an interpreter, or justify which to use (9 questions)

1. **Note the stage**: while writing or testing an unfinished program, or when the program is finished and sold.
2. **Choose**: interpreter while writing, compiler when finished (repeated testing of finished code is also a compiler point). Some questions give no mark for the choice itself, only for the reasons.
3. **Give reasons tied to that stage**, each with its "so": "stops at the first error, **so** it can be corrected in real time and the program continued from there". Describing how the translator works earns nothing here.
4. **Compare**: the question says "compared to", so each point must say what the other translator does or can't do.

Variations you will meet:

- **Drawbacks of a compiler during development**: slow to compile large programs; must recompile after every change; won't run with any errors present; one error can cause a list of false follow-on errors; unfinished code can't be tested.
- **"Why is debugging easier with an interpreter?"** or **"why a compiler for the finished product?"**: use the two lists under "Which to use when" in "Understand it", each point with its "so".
- **"How can the programmer use both?"**: interpreter while coding (test and fix as you go), then compiler at the end (executable, repeated tests, distribution).

5. Name the OS key management tasks or state what an OS is for (8 questions)

1. **Use the five syllabus names**: memory, file, security, hardware, process management.
2. **Don't repeat** the task the question gives you, and don't give one task twice under two names ("hardware management" and "device management" are the same).
3. **"State one purpose of an OS"**: provides a user interface; hides the complexity of the hardware; provides a platform for software to run; manages the computer's resources.

Variations you will meet:

- **Matching tasks to descriptions**: memory → allocates memory to processes; security → checks users and processes are genuine; hardware → installs drivers for devices; process → lets processes pass data to each other; file → keeps track of free storage.
- **"Describe the purpose of an OS"** for 5 marks: give a task, then "so" what it achieves, several times, for example "manages processes, so each gets fair access to the processor".

6. Describe what one OS management task does (8 questions)

1. **Identify exactly which task**, and remember that "process" means a running program and "memory" means RAM.
2. **Write the separate things that task does** (the bullets in "Understand it"), one mark each, adding a "so" where it helps.

Variations you will meet:

- **Process management** is the one most often answered wrongly: describe what happens to running programs (scheduling, multitasking, fair access), **not** a list of all the OS's tasks.
- **Memory versus file management**: RAM (allocating memory to processes, keeping running programs apart) versus storage (file system, folders, file operations, access rights). Examiners see these mixed up every year.
- **How memory and process management support multitasking**: use the paragraph at the end of "The five key management tasks" in "Understand it", giving points from both tasks.
- **Hardware or security management** (often "state two tasks"): pick two separate bullets from that task in "Understand it", such as installing drivers and handling interrupts, or authentication and access rights.

7. Describe how a compiler, interpreter or assembler works (4 questions)

1. Describe **what it does**, in order, not its benefits.
2. Follow the steps in "How a compiler works" or "How an interpreter works" in "Understand it"; an assembler translates assembly language into machine code.

Variations you will meet:

- **Fill the gaps** in a paragraph. The missing words are single key terms: "executable" (file), "source code", "stops", "immediately" or "in real time", "machine code". Read the whole sentence before choosing, and don't use a compiler word in an interpreter sentence.

8. Explain why a language is partly compiled and partly interpreted (2 questions)

1. Say what happens: compiled to **intermediate code** (bytecode), then **interpreted** on each computer by a virtual machine.
2. Give the reason: the same intermediate code runs on **different platforms** without recompiling the source code; it can also be optimised for the processor running it.

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 smartphone has an Operating System.

(a) Other than process management, identify **three** key management tasks carried out by the Operating System. **[3]**

(b) The user stores photos, music and documents on the phone. Describe how the Operating System manages these files. **[3]**

(c) The user connects the phone to a wireless printer. Describe how the Operating System's hardware management allows the phone to print a document. **[3]**

(a) Any three of: memory management, file management, security management, hardware management. **B1 B1 B1**

(b) Any three points, one mark each:

- It sets up a file system that divides storage into blocks and records which blocks each file uses and which are free. **B1**
- It keeps a folder (directory) structure, so photos, music and documents can be organised and found. **B1**
- It lets the user create, open, copy, move, rename, delete and search for files. **B1**
- It controls access to each file, for example which apps may read the photos.

(c) Any three points:

- It installs and uses a **device driver** for the printer, which turns the OS's print instructions into commands this printer understands. **B1**
- It **sends the data** to the printer and receives signals back, such as "out of paper". **B1**
- It holds the document in a **buffer** (and a print queue), because the printer works far more slowly than the processor. **B1**
- It handles the printer's **interrupts**, for example when it needs more data.

Answers about process management or the user interface score nothing in (c): the question is about hardware management.

Example 2

An office computer stores its files on a magnetic hard disk. Opening files has become slower over time.

(a) Identify the utility software that could improve this. Explain how it improves the performance of the computer. **[4]**

(b) The office also has a laptop with a solid state drive. Explain why this utility is not needed on the laptop. **[2]**

(c) Describe how back-up software protects the office's data. **[2]**

(a) Defragmentation software. B1

- Over time, as files are saved and deleted, files are split into fragments in different places on the disk; the software moves them so that each file is stored in contiguous blocks. **B1**
- So the read/write head needs to move less to read a file. **B1**
- So each file is accessed faster, and the computer runs faster. **B1**

A description of how the software rearranges blocks, with no link to speed, gets only the first two marks.

(b) A solid state drive has no moving read/write head, so reading scattered blocks takes (almost) no extra time. **B1** Moving the data would only add extra writes, which wear the drive out. **B1**

(c) It makes copies of the files on another device or location, regularly and automatically, so no one has to remember. **B1** If the original files are lost or corrupted, the copies are used to restore them. **B1**

Example 3

A team is writing a route-planning app in a high-level language. Their IDE has both an interpreter and a compiler.

(a) Explain why the team uses the interpreter while they are writing the code. **[3]**

(b) Explain why they use the compiler when the app is finished and ready to sell. **[3]**

(c) Their language is compiled into intermediate code, which is then interpreted when it runs. State **two** reasons why a language is designed to work this way. **[2]**

(a) Any three points, linked to writing the code:

- The interpreter stops at the first error it meets. **B1**
- So the team can correct that error straight away and continue running from that line, seeing the effect of the change immediately. **B1**
- Errors appear one at a time, so there are no knock-on errors to confuse them. **B1**
- Parts of the app can be tested before the rest is written.

(b) Any three:

- The compiler produces an executable file. **B1**
- So customers can run the app without having the translator. **B1**
- The source code is not given out, so others cannot read, change or copy it. **B1**
- The executable can be run and tested many times without being translated again.

(c) Any two: the intermediate code runs on different platforms, each with its own virtual machine **B1**; the source code does not have to be recompiled for each kind of computer **B1**; the code can be optimised for the processor it runs on.

"The interpreter is used first, then the compiler" answers a different question and earns nothing in (c).

Example 4

A student's program calculates the average of a set of test marks, but prints the wrong average. She is using an IDE.

(a) Describe how she can use **two** debugging features of the IDE to find the error. **[4]**

(b) Her teacher suggests keeping useful routines in a Dynamic Link Library (DLL) rather than an ordinary library. Explain **two** benefits of using a DLL. **[4]**

(c) State **one** drawback of using a library routine written by someone else. **[1]**

(a) One mark for each feature, one for describing its use:

- **Breakpoint:** **B1** she sets one on the line that works out the average, so the program runs up to there and pauses, and she can check the total and the count at that point. **B1**
- **Single stepping with a watch window:** **B1** she runs the loop one line at a time and watches the total and counter change after each line, to see where a value goes wrong. **B1**

(b) One mark for each benefit, one for its matching expansion:

- The DLL can be updated on its own; **B1** programs that use it get the change without being recompiled. **B1**
- Less main memory is used; **B1** the DLL is loaded only when it is needed, and one copy in memory is shared by every program that uses it. **B1**

(c) Any one: it may not do exactly what she needs; it may not be compatible with her program; it may contain bugs she can't see; a later change by its author may make her program behave differently. **B1**

Common mistakes

- **Describing the thing instead of answering the question.** The most common problem in this topic. "Why use an interpreter while coding?" needs reasons tied to coding, not how an interpreter works; "how does defragmentation improve performance?" needs the link to faster access. Read the question twice and finish every point with how it answers it.

- **Mixing up memory and storage.** Memory management is about RAM; file management is about the disk. "Saves storage" and "saves memory" are different DLL points.
- **Vague words.** "Faster", "easier", "saves space" and "saves time" earn nothing alone. Say what is faster, which space, and why.
- **General library benefits for a DLL question.** A DLL question wants DLL-only points: loaded when needed, shared in memory, smaller executable, no recompiling.
- **Swapping IDE features between groups.** Context-sensitive prompts and prettyprint are for coding and presentation, not debugging. Context-sensitive prompts suggest code; dynamic syntax checks highlight errors.
- **Repeating the name as the description.** "Single stepping steps through the code" earns nothing; "runs one line at a time so the effect of each line can be seen" does.
- **Defragmentation as compression.** Defragmentation makes each file's fragments **contiguous**; it does not make files smaller.
- **Treating "process" as "any job the OS does".** In process management, a process is a running program: talk about scheduling and multitasking.
- **Repeating a task you were given,** or giving one task twice under two names.
- **Answering "partly compiled, partly interpreted" with "interpreter first, compiler later".** It is about intermediate code interpreted on different platforms.

How to get the marks

- **One point per mark.** Count the marks and write at least that many separate, different points. Put each on its own line.
- **Point, then "so".** Many questions pair a point with its expansion ("pre-tested ... **so** less testing time"). A bare list often gets half the marks.
- **"State", "Identify", "Give", "Name":** a short answer is enough, but it must be the proper term (breakpoint, defragmentation, memory management).
- **"Describe":** say what it is or does, in several points. **"Explain":** give reasons, how or why, with "because" or "so". **"Justify":** explain why your choice suits this situation; the choice itself may earn nothing.
- **"Compared to":** every point needs both sides, or a clear "whereas".
- **Use the context.** If the question is about a team, a phone or a game for sale, tie your points to it.
- **"Each feature must be different":** one feature can't score twice, even in two rows.
- **Gap fills** need the exact term; a nearly-right word usually scores nothing.

Quick check

1. State **two** reasons why a computer needs an Operating System.
2. Name the utility that: (a) scans files for malicious code; (b) sets up a file system on a new disk; (c) reduces the size of a file before it is emailed.

3. Name the IDE feature that: (a) underlines a misspelt keyword as it is typed; (b) hides the body of a finished procedure so the rest of the code can be seen; (c) pauses the program each time it reaches line 40.
4. Give **two** drawbacks of distributing a finished program to customers as source code to be interpreted, compared with a compiled executable.
5. A programmer writes a device driver in assembly language. State what an assembler does, and why the assembly program must be translated before it runs.

Answers

1. Any two: it provides a user interface; it hides the complexity of the hardware from the user and programs; it provides a platform for applications to run; it manages the computer's resources (memory, files, security, hardware and processes).
2. (a) Virus checker. (b) Disk formatter. (c) File compression.
3. (a) Dynamic syntax check. (b) Collapsing a code block (expand/collapse). (c) Breakpoint.
4. Any two: each customer needs the interpreter; the program is translated every time it runs, so it runs more slowly; the source code is handed over, so it can be read, changed or copied.
5. An assembler translates assembly language (mnemonics) into machine code. The processor can only execute machine code, so the assembly program cannot run until it has been translated.

Past questions to try

- **9618/13/M/J/22 Q4**: describe the OS key management tasks, then the purpose of back-up and defragmentation software, and one other utility.
- **9618/12/O/N/22 Q1**: match five utilities to six purposes, two drawbacks of a compiler during development, and why programs may be partly compiled and partly interpreted.
- **9618/12/M/J/23 Q7**: define a program library and two benefits of creating a DLL, choose and justify a translator during development, and describe four IDE features.