Course Companion

# A-Level Computer Science

Every topic handout in one volume

全部专题讲义合集

exercises.lol

## Contents 目录

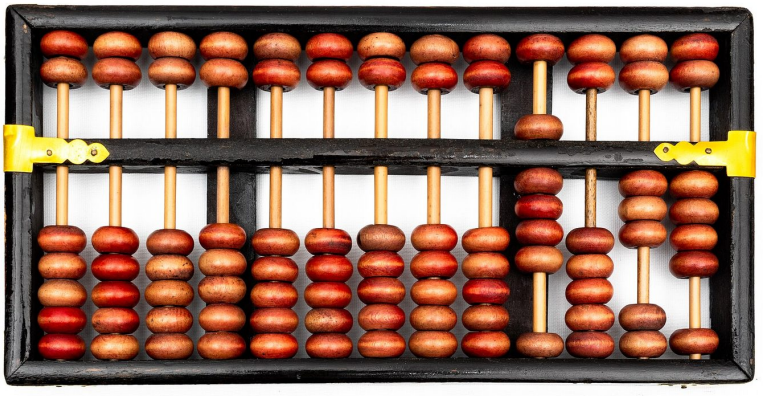
# Information representation

A-Level Computer Science

## Number systems

The three **number systems** 数制 you must use:

- **denary** 十进制 (decimal, base 10) — uses digits 0–9. Place values are powers of ten.
- **binary** 二进制 (base 2) — uses 0 and 1. Place values are powers of two. Every **byte** 字节 is 8 **bits** 位.
- **hexadecimal** 十六进制 (base 16) — uses 0–9 then A–F for 10–15. Each hex **digit** 数位 stands for exactly 4 bits.



*An abacus represents numbers by place value — the same idea behind decimal, binary and hexadecimal*

### Conversions

**Denary → binary**: keep dividing by 2 and record the remainders, read bottom-up. Or subtract the largest **place value** 位值 (power of 2) that fits.

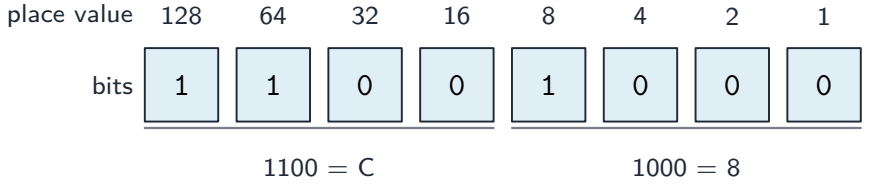
Example: $558_{10}$: $558 = 512 + 32 + 8 + 4 + 2 = 2^9 + 2^5 + 2^3 + 2^2 + 2^1$. In 12 bits: `0010 0010 1110`.

**Binary → hex**: group the bits into **nibbles** 半字节 (4 bits) from the right and convert each. `0010 0010 1110` → `2 2 E` → `22E`.

**Hex → binary**: replace each hex digit with its 4-bit pattern. **Hex → denary**: multiply each digit by its place value. `22E` $= 2 \times 256 + 2 \times 16 + 14 = 558$.

**Worked example.** Convert denary 200 to 8-bit binary, then to hexadecimal.

$200 = 128 + 64 + 8$, so the binary is `11001000`. In nibbles, `1100 1000` = 12 and 8, i.e. C and 8, so the hexadecimal is `C8`.




$$200 = 128 + 64 + 8 = \texttt{11001000}_2 = \texttt{C8}_{16}$$

*Reading 200 from its place values, then grouping the bits into nibbles to get hex C8*

## Binary vs decimal prefixes

Two prefix families look similar but differ — decimal (powers of 10) and binary (powers of 2):

| Decimal (SI) | Binary (memory) |
|---|---|
| kilo $= 10^3$ | kibi (Ki) $= 2^{10} = 1024$ |
| mega $= 10^6$ | mebi (Mi) $= 2^{20}$ |
| giga $= 10^9$ | gibi (Gi) $= 2^{30}$ |
| tera $= 10^{12}$ | tebi (Ti) $= 2^{40}$ |

So a tebibyte (TiB) is slightly more than a terabyte (TB). A "1 TB" drive holds $10^{12}$ bytes, but an operating system that reports in TiB shows a smaller number.

# Binary arithmetic

## Binary addition

Add column by column from the right, carrying as in denary:

| Bit A | Bit B | Carry in | Sum bit | Carry out |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 0 |
| 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | 0 | 1 |
| 1 | 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 1 | 1 |

**Overflow** 溢出 happens when the result needs more bits than the **register** 寄存器 can hold — the carry-out of the leftmost column is the overflow bit.

## Binary subtraction

The usual way is **two's complement** 补码 addition: to do $A - B$, form the two's complement of $B$ (invert every bit and add 1), then add, and discard any final carry-out.

To subtract 00011110 from 01100100 (unsigned 8-bit):

- two's complement of 00011110: invert $\rightarrow$ 11100001, add 1 $\rightarrow$ 11100010.
- add to 01100100: result 1 01000110 (9 bits) — discard the leading 1 $\rightarrow$ 01000110 = $70_{10}$. Check: $100 - 30 = 70$. ✓

## Two's complement signed integers

In an $n$-bit two's-complement number:

- the **most significant bit** 最高有效位 (MSB) is the **sign bit** 符号位: 0 = positive, 1 = negative.
- to read a negative number: invert every bit, add 1, then negate.

So 11100010 is negative; invert $\rightarrow$ 00011101, add 1 $\rightarrow$ 00011110 = 30, so it is $-30$. This is a **signed integer** 有符号整数 (unlike an **unsigned** 无符号 one). The range for $n$ bits is $-2^{n-1}$ to $+2^{n-1} - 1$; for 8 bits, $-128$ (10000000) to $+127$ (01111111).

The same bits mean different numbers depending on the agreed reading. As an **unsigned** integer every bit is a place value, so 8 bits run from 0 to 255; as a **signed** two's-complement integer the top bit is the sign, so the same 8 bits run from $-128$ to

+127. The pattern 11111111 is 255 read one way and −1 read the other — nothing in the bits themselves says which.

| 8 bits | as unsigned<br>every bit a place value | as signed<br>top bit is the sign |
|---|---|---|
| 00000000 | 0 | 0 |
| 01111111 | 127 | +127 |
| 10000000 | 128 | −128 |
| 11111111 | 255 | −1 |
| range for $n$ bits | $0 \dots 2^n - 1$ | $-2^{n-1} \dots 2^{n-1} - 1$ |

*The same byte read as unsigned and as signed: only the agreed interpretation tells them apart*  *8-bit two's complement: the sign bit splits the range into negative (−128 to −1) and positive (0 to 127)*

**Worked example.** What denary value does the 8-bit two's-complement number 10110100 represent?

The MSB is 1, so it is negative. Invert → 01001011, add 1 → 01001100 = 76, so the value is −76. Check with place values: $-128 + 32 + 16 + 4 = -76$.

Overflow in signed arithmetic happens when the true result falls outside this range — spotted when the sign bit flips wrongly (two positives giving a negative, or two negatives giving a positive).

## One's complement

Before two's complement, an older scheme called **one's complement** 反码 represented a negative number by simply **inverting every bit** of the positive — there is no "add 1" step.

- +30 = 00011110, so in one's complement −30 = 11100001 (just the inverse).
- Drawback: it has **two zeros** — 00000000 (+0) and 11111111 (−0) — which wastes a bit pattern and makes arithmetic awkward.

**Two's complement** (invert **and** add 1) removes the negative zero: it has a single zero and lets addition and subtraction use the same circuit. That is why modern computers store signed integers in two's complement, not one's complement.

# Binary Coded Decimal (BCD)

In **BCD** 二进码十进数, each denary digit is written as its own 4-bit pattern. The number 93 is `1001 0011` in BCD — not binary 93 (01011101). Each nibble uses only 0–9; patterns 1010–1111 are invalid.

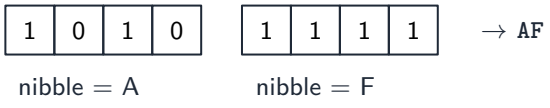
BCD reading: `0010 0111 0101` → 2, 7, 5 → 275.

**Use**: calculators, digital clocks, and devices that show denary digits — each digit drives a **7-segment display** 七段显示器. Currency code often uses BCD to avoid the rounding errors of converting fractions like 0.1 to binary.

*A seven-segment display shows one denary digit, often driven by BCD*

# Hexadecimal — practical uses

Hex is a compact way to write binary (1 hex digit = 4 bits):




*A byte is two nibbles; each nibble is one hex digit*

- **memory addresses** 内存地址 in low-level programming — `0x7FFE`.
- **colour values** in HTML/CSS — `#FF8800`.
- **MAC addresses** — `AC:DE:48:00:11:22`.

Hex does not change the stored data — it just makes binary easier for humans.

# Character codes

Computers store text as numbers; each character has a numeric **code point** 码点 set by a **character set** 字符集.

## ASCII

- **ASCII** uses 7 bits — 128 code points. Basic Latin letters, digits, punctuation, and control codes.
- **Extended ASCII** uses 8 bits — 256 code points; the lower 128 match ASCII, the upper 128 vary by region.

| character | code (denary) | binary |
|---|---|---|
| A | 65 | 01000001 |
| a | 97 | 01100001 |
| 0 | 48 | 00110000 |
| (space) | 32 | 00100000 |

*Each character is stored as a number — a few ASCII code points in denary and binary*

## Unicode

- **Unicode** is a universal character set covering almost every script, plus symbols and emoji.
- common **encodings** 编码: **UTF-8** (1–4 bytes, ASCII-compatible), **UTF-16** (2 or 4 bytes), **UTF-32** (fixed 4 bytes).

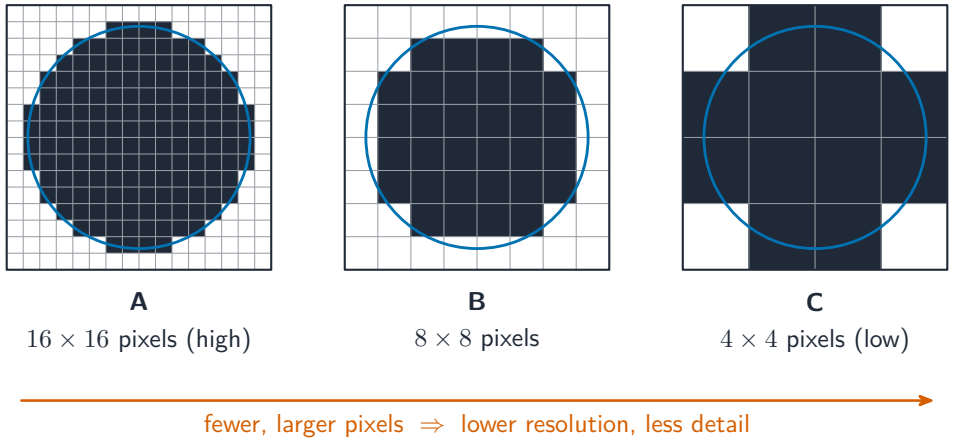
## Why Unicode beats ASCII

- it represents **far more characters** (every script, emoji); ASCII covers only basic English.
- files are **portable** with no code-page confusion, and allow **multilingual** text in one document.
- trade-off: Unicode files are usually **larger** for English-only text.

# Bitmap images

A **bitmap** 位图 image (also called a **bitmapped image**) stores the colour of every **pixel** 像素 in a grid. At the start of the file a **file header** 文件头 records the image's metadata — its width, height and colour depth — so software knows how to read the pixel data that follows.

- **image resolution** 图像分辨率: the bitmap's own size, width × height in pixels (e.g. 1920 × 1080).
- **screen resolution** 屏幕分辨率: the width × height the **display** can show. If an image's resolution is larger than the screen it is scaled down to fit; a low-resolution image looks blocky when stretched onto a higher-resolution screen.
- **colour depth** 颜色深度 (**bit depth** 位深度): bits per pixel. 1 bit → black/white; 8 bits → 256 colours; 24 bits → 16.7 million ("true colour").




*The same image stored at three resolutions, from high (A) to low (C): fewer, larger pixels mean less detail*

## File size

$$\text{size in bits} = \text{width} \times \text{height} \times \text{bit depth}.$$

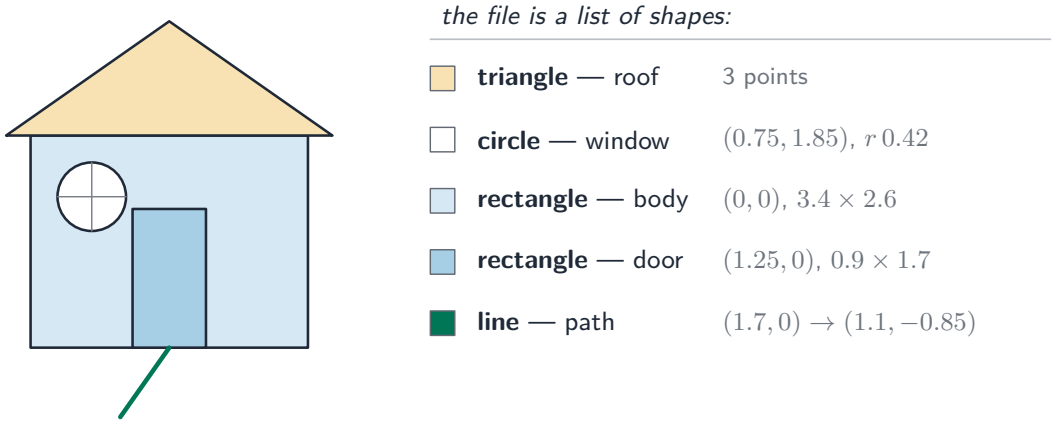
Divide by 8 for bytes, by 1024 for KiB, etc. Example: a 3000 × 2000 image at 24 bpp is $3000 \times 2000 \times 24 = 1.44 \times 10^8$ bits ≈ 17.2 MiB.

### Changing settings

- **lower resolution** → smaller file, less detail (looks blocky when enlarged).
- **lower colour depth** → smaller file, but smooth shades show banding.
- **higher** of either → larger file, better quality.

## Vector graphics

A **vector graphic** 矢量图形 stores the **instructions** to draw the image as a **drawing list** 绘图列表 — an ordered list of **drawing objects** 绘图对象 (geometric **primitives** 图元: lines, curves, polygons, circles). Each drawing object has **properties** 属性 such as colour, fill, line width and position (coordinates). To show it, the program **renders** 渲染 the drawing list at any resolution needed.

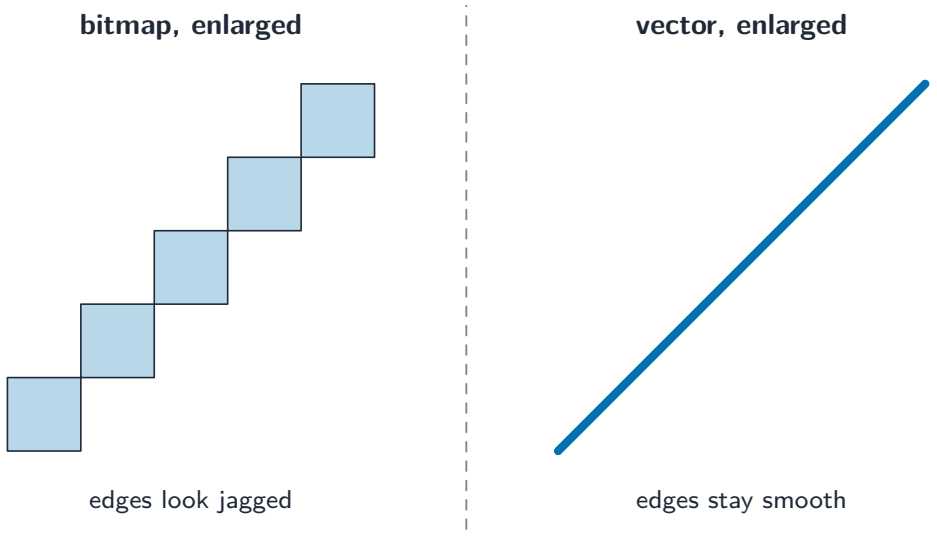



*A vector image is built from labelled geometric shapes, each with attributes*

### Bitmap vs vector

| Task | Better choice | Why |
|---|---|---|
| Photograph | Bitmap | Complex pixel-level detail can't be described as shapes. |
| Logo, icon, sign | Vector | Sharp edges; scales to any size without blur. |
| Engineering drawing | Vector | Precise geometry and scaling. |
| Painting, texture | Bitmap | Smooth tonal detail per area. |

**Vector advantage**: it **scales without losing quality** — a vector logo stays sharp at any size, while a bitmap blurs when enlarged. **Vector disadvantage**: it cannot describe arbitrary pixel detail (photographs).

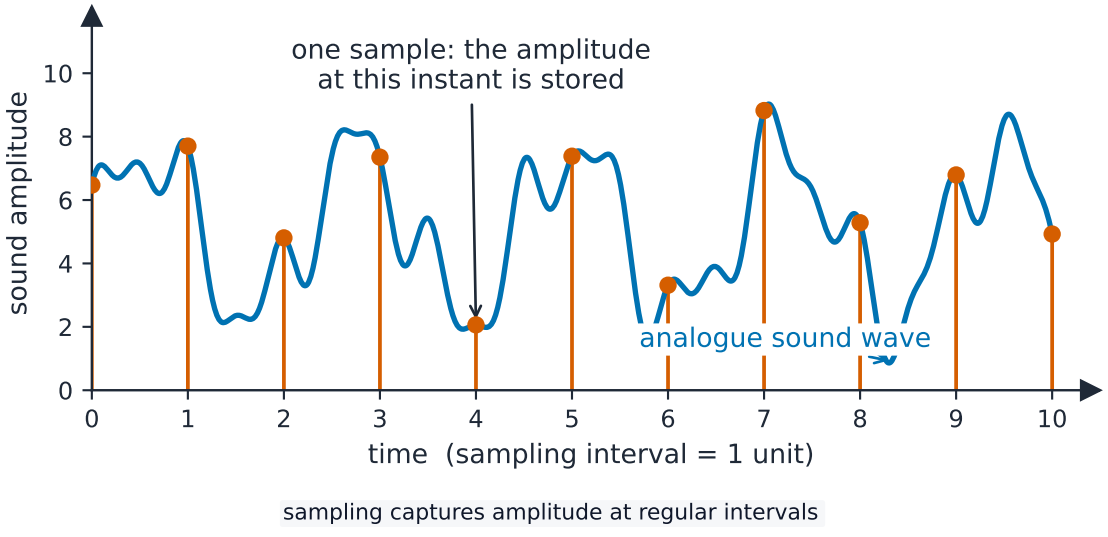



*Enlarged, a bitmap's pixels turn jagged; a vector stays smooth at any size*

# Sound

A continuous wave of **analogue data** 模拟数据 (the sound) is converted into **digital data** 数字数据 by **sampling** 采样:

- **sampling rate** 采样率 — samples per second (Hz). CD quality is 44.1 kHz.
- **sampling resolution** 采样分辨率 (bit depth) — bits per sample's **amplitude** 振幅. CD quality is 16 bits.




*Sampling a sound wave: its amplitude is read at each time interval*

### File size

$$\text{size in bits} = \text{sampling rate} \times \text{resolution} \times \text{duration} \times \text{channels}.$$

A 10-second stereo CD clip: $44100 \times 16 \times 10 \times 2 = 14\,112\,000$ bits $\approx 1.68$ MiB.

### Changing settings

- **higher sampling rate** → captures higher pitches, larger file.
- **higher sample resolution** → finer amplitude steps, less **quantisation** 量化 noise, larger file.
- **lower** of either → smaller file, clear quality loss.

(The sampling rate must be at least twice the highest frequency you want to keep.)

## Compression

**Compression** 压缩 reduces file size, saving storage and transmission **bandwidth** 带宽. Two kinds:

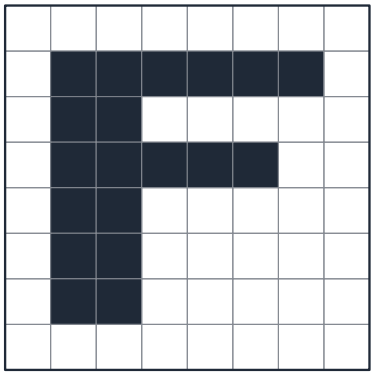
- **lossless** 无损 — the original data is recovered exactly (text, programs, ZIP/PNG).
- **lossy** 有损 — some detail is dropped for much smaller files (JPEG, MP3, video).

### When to use which

- **lossless** for documents, source code, medical images — anything needing exact data.
- **lossy** for streaming media. **Real-time video streaming** uses lossy compression because it must send huge amounts of data in real time over limited bandwidth; lossless would not shrink it enough. Raw HD video is gigabytes per minute, so without compression the picture would keep freezing.

### Lossless methods

- **run-length encoding** 行程编码 (RLE): store "the next $n$ values are $x$" instead of repeating $x$. Great for flat areas; useless for noisy data.
- **dictionary methods** 字典编码 (ZIP, PNG): replace repeated byte sequences with a short reference. Good for text and code.
- **Huffman coding** 霍夫曼编码: give short codes to common symbols and long codes to rare ones, bringing the average code length near the data's **entropy** 熵.

| 8×8 bitmap | binary | RLE code |
|---|---|---|
|  | 00000000 | 8W |
| | 01111110 | 1W 6B 1W |
| | 01100000 | 1W 2B 5W |
| | 01111100 | 1W 5B 2W |
| | 01100000 | 1W 2B 5W |
| | 01100000 | 1W 2B 5W |
| | 01100000 | 1W 2B 5W |
| | 00000000 | 8W |

**Key:** 1 = black (B), 0 = white (W). RLE stores *count* + *colour*: e.g. 01111110 → 1W 6B 1W

*Run-length encoding of the letter F in an 8 × 8 black-and-white grid*

**source** `ABC` `ABC` `ABC` `XYZ` 12 characters

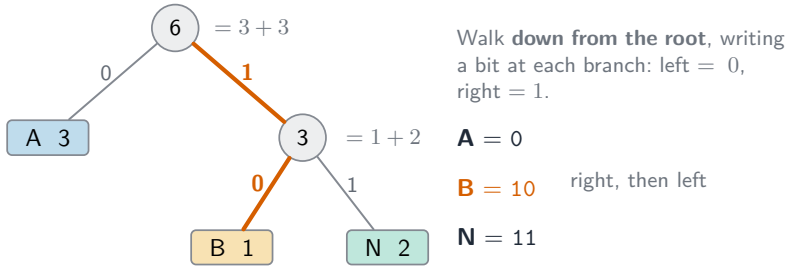
**dictionary** `1` → `ABC` `2` → `XYZ` stored once each

**encoded** `1` `1` `1` `2` 4 tokens + dictionary

---

The decoder rebuilds the dictionary as it reads, so the original returns exactly — **lossless** (ZIP, PNG).

*Dictionary coding: each repeated sequence is stored once, and every occurrence becomes a short index*

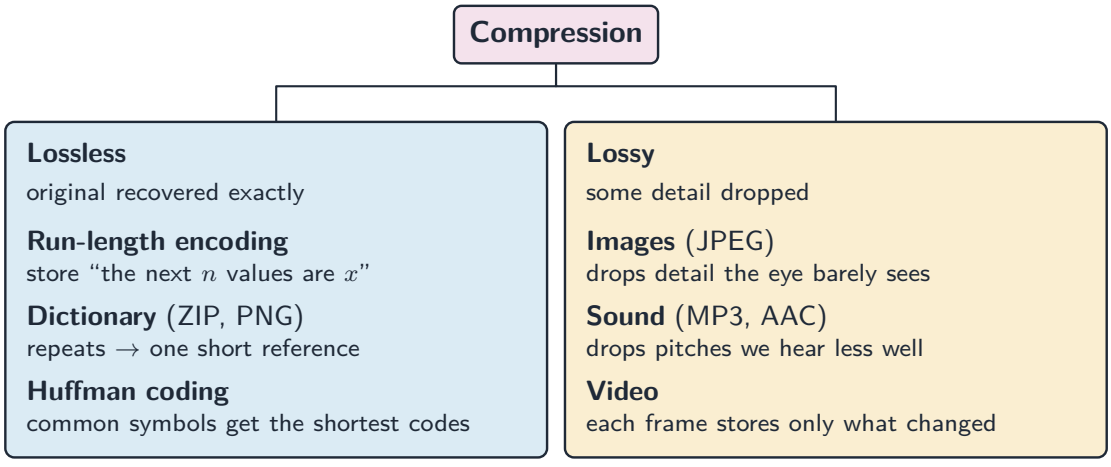



BANANA becomes 10 0 11 0 11 0 — **10 bits**, against 12 for fixed 2-bit codes: the commonest letter is nearest the root.

*Huffman coding: the commonest symbol gets the shortest code, so BANANA needs 10 bits instead of 12*

## Lossy methods

- **images** (JPEG): drop fine detail and colour differences the eye barely sees.
- **sound** (MP3, AAC): drop pitches we hear less well, and quiet sounds hidden by louder ones.
- **video** combines **spatial** 空间 compression (within each frame, like JPEG) with **temporal** 时间 compression (most frames store only the differences from the previous frame).




*Compression methods: lossless versus lossy, with common examples*

# Exam tips

- Show working for base conversions: denary → binary by place values, binary → hexadecimal in **nibbles** (groups of 4 bits).
- For **two's complement** the MSB is negative; to negate, **invert and add 1**; watch for **overflow** when the sign bit flips wrongly.
- Distinguish **bitmap** (pixels; file size = width × height × colour depth) from **vector** (drawing commands; scales without loss).
- Sound file size depends on **sample rate** × **bit depth** × **time** — more of each means better quality but a bigger file.
- Compare **lossless vs lossy** compression and give a use for each.

# Communication

A-Level Computer Science

## Networks: purpose and benefits

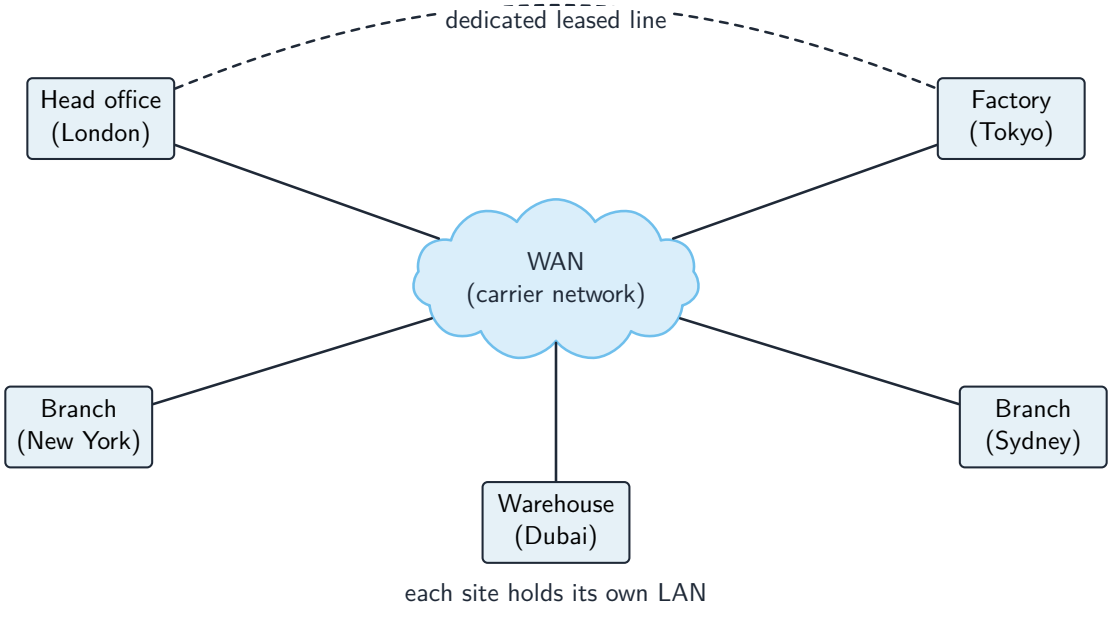
A **network** 网络 is a set of computing devices connected so they can communicate and share resources. Benefits:

- **sharing resources** (printers, file servers, internet) — cheaper than equipping each computer.
- **sharing data** — many users access the same files.
- **central management** — install software, manage users and back up once on a server.
- **communication** — email, video calls, messaging.
- **remote access** — work from anywhere.

# LAN vs WAN

A **local area network** 局域网 (LAN) covers a **small area** — a home, office or school, usually owned by the organisation, with high data rates and low **latency** 延迟.

A **wide area network** 广域网 (WAN) covers a **large area** — a city, country, or the world (the internet is the largest WAN). It uses telecom-company infrastructure — often the **Public Switched Telephone Network** 公共交换电话网 (PSTN), leased lines or fibre — with lower data rates and higher latency. A WAN connects LANs together.

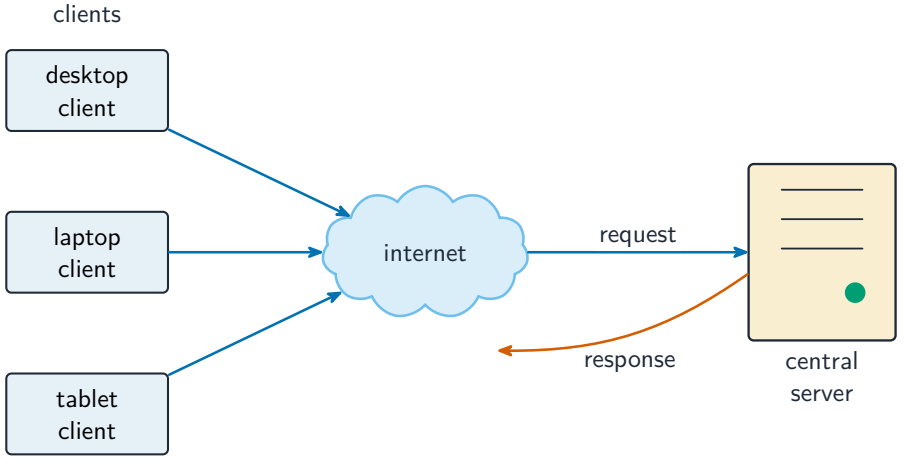



*A wide-area network links many systems across a large area*

# Client-server and peer-to-peer

## Client-server

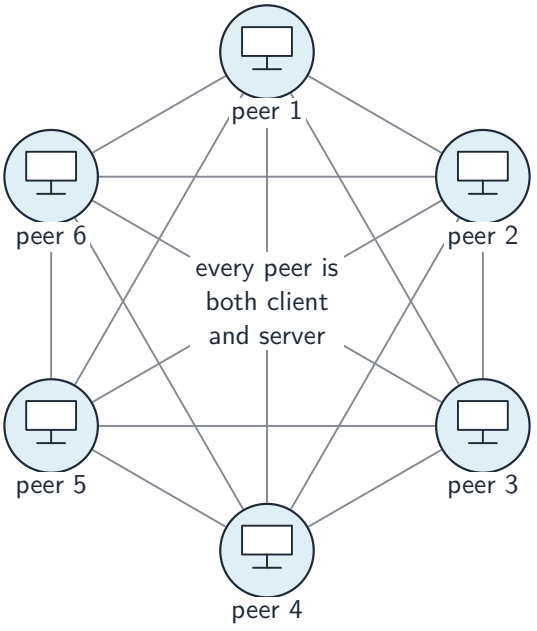
- powerful machines act as **servers** 服务器, providing services (files, web pages, email).
- other machines are **clients** 客户端 that request services.
- central and easy to manage, but the server is a single point of failure unless backed up.




*In a client-server network, clients request services from a central server*

### Peer-to-peer (P2P)

- all machines are equal **peers**; each can be both client and server (**peer-to-peer** 对等网络).
- resources are spread across the peers — no central server. Robust to one failure, but harder to keep secure and consistent.




*In a peer-to-peer network, every node is both client and server*

## Thin and thick clients

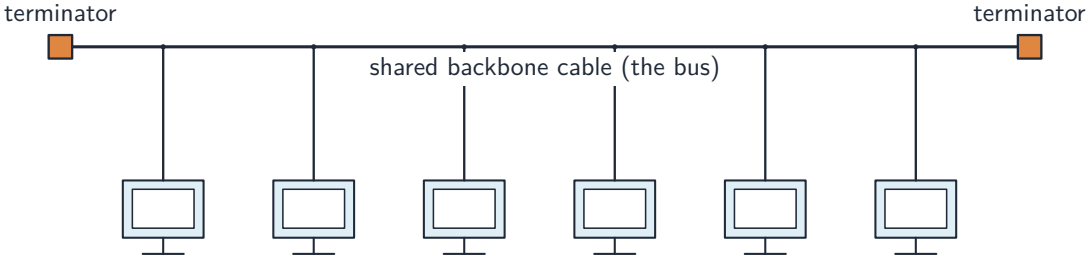
A **thin client** 瘦客户端 does **little processing locally** and relies on a powerful server (web terminals, remote desktops). A **thick client** 胖客户端 has **strong local processing and storage** and runs full applications itself (a normal desktop PC).

| Feature | Thin client | Thick client |
|---|---|---|
| Local processing | minimal | substantial |
| Local storage | minimal | substantial |
| Reliance on network | high | lower |
| Server load | high | lower |

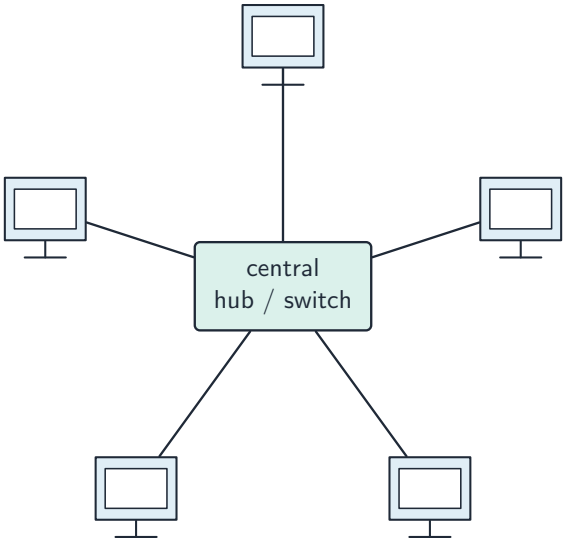
# Network topologies

The **topology** 拓扑 is how the nodes and links are arranged.

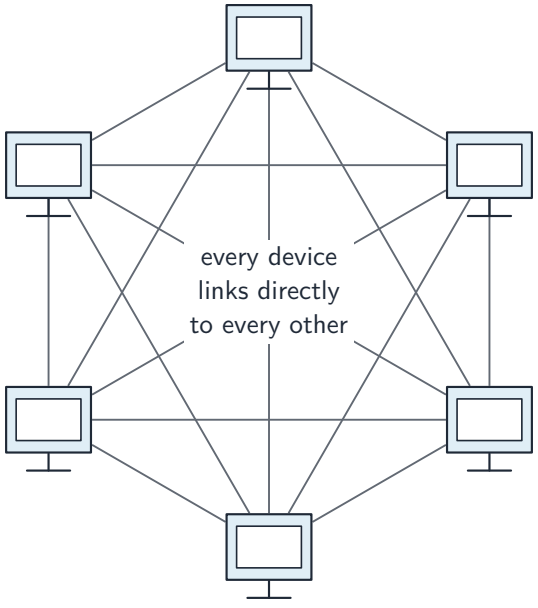
- **bus** 总线 — all devices on one shared cable. Cheap; the whole LAN fails if the bus fails; performance drops as more devices share the **bandwidth** 带宽.
- **star** 星形 — every device connects to a central switch. One device failing does not affect others; the switch failing brings all down. Most common today.
- **mesh** 网状 — every device links directly to others, with many paths. Very **fault-tolerant** 容错 (traffic reroutes) but needs lots of cabling.
- **hybrid** — a mix (a star in each office, mesh links between offices).



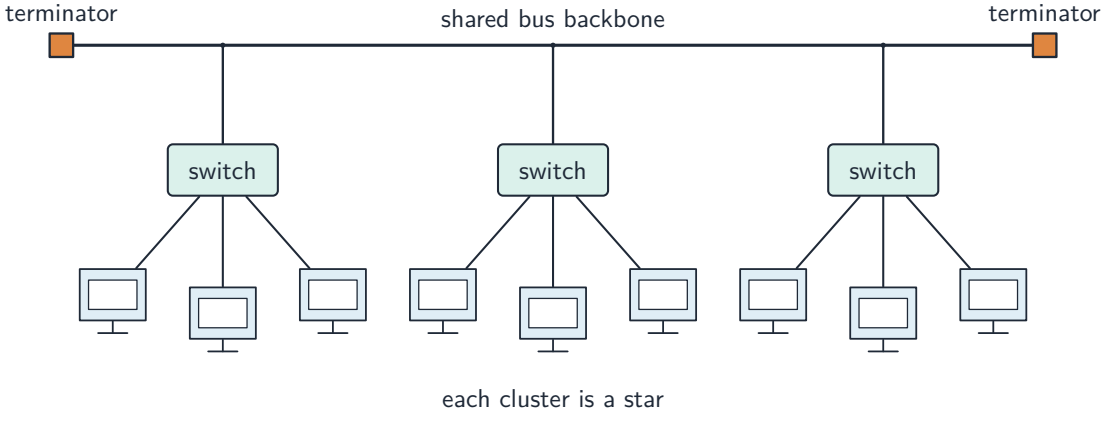

*Bus topology: all devices share one cable with a terminator at each end*




*Star topology: every device connects to a central hub or switch*




*Mesh topology: every device links directly to the others*




*Hybrid topology: star clusters joined by a central bus*

# Cloud computing

**Cloud computing** 云计算 delivers computing services (servers, storage, software) **over the internet**, hosted by a third party. Benefits: **scalability** 可扩展性 (pay for what you need), lower cost, access from anywhere, and reliable redundant data centres. Drawbacks: needs internet, your data is held by a third party, and possible vendor lock-in.

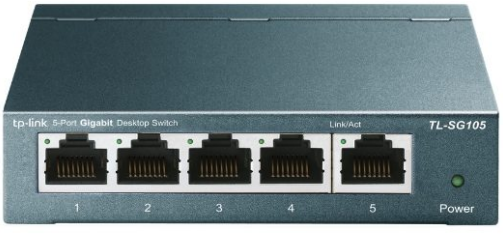
## Wired vs wireless

- **wired** (**Ethernet** 以太网 over **twisted-pair** 双绞线 or **fibre-optic** 光纤): higher speed, lower latency, fewer errors, more secure.
- **wireless** (Wi-Fi, Bluetooth, cellular): no cables, devices can move, but slower, prone to interference and eavesdropping.

For the same generation, wired wins on speed and reliability; wireless wins on convenience.

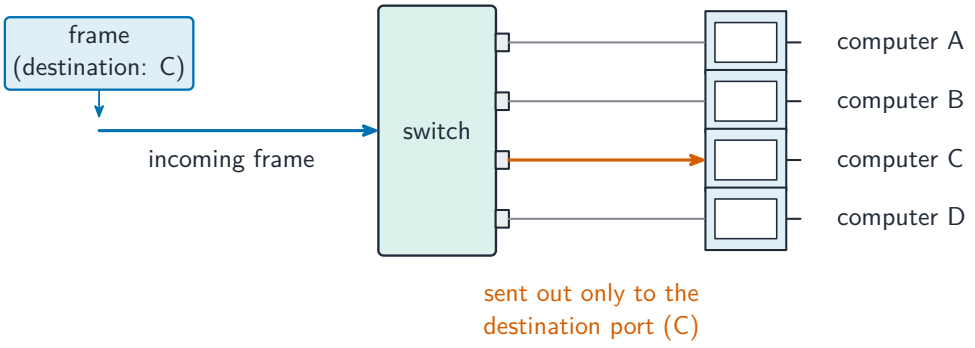
## LAN hardware

- **network interface card** 网络接口卡 (NIC) — lets a device send and receive on the network; has a unique **MAC address** MAC 地址 (a 48-bit hardware address). A wireless device uses a **wireless network interface card** 无线网络接口卡 (WNIC).
- **switch** 交换机 — forwards Ethernet frames only to the port for the destination MAC address.
- **hub** 集线器 — a simpler device that copies traffic to all ports (now obsolete).
- **wireless access point** 无线接入点 (WAP) — lets wireless clients join a wired LAN.
- cabling — twisted-pair for short runs; fibre-optic for longer, faster runs.




*A network switch: each device's cable plugs into one of its ports*

*An RJ-45 plug on a twisted-pair Ethernet cable*

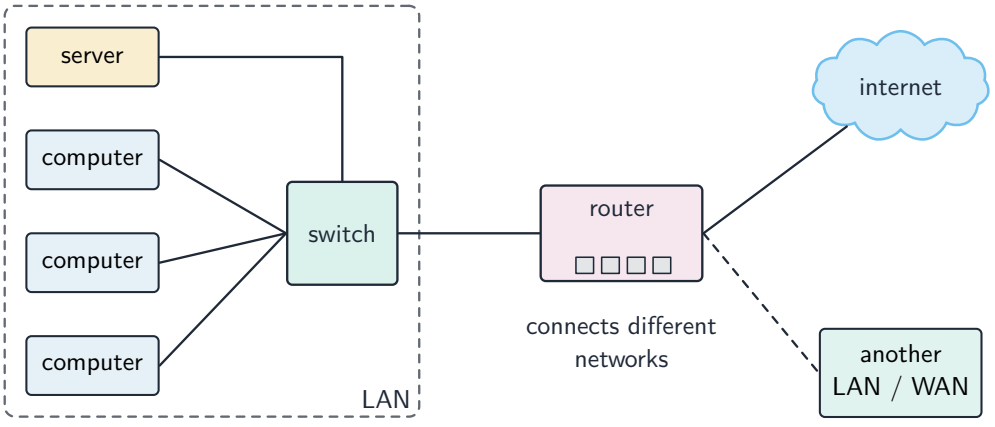



*A switch sends each frame only to the port for its destination*

# Routers

A **router** 路由器 connects **different networks** and forwards data between them — usually at the boundary of a LAN and the internet. It does:

- **forwarding** — reads each **packet** 数据包's destination **IP address** IP 地址 and sends it out the right port, using a **routing table** 路由表.
- **network address translation** 网络地址转换 (NAT) — lets many private LAN addresses share one public IP.
- **DHCP** 动态主机配置协议 — hands out private IP addresses to LAN devices.
- **firewall** 防火墙 — blocks unwanted incoming traffic.



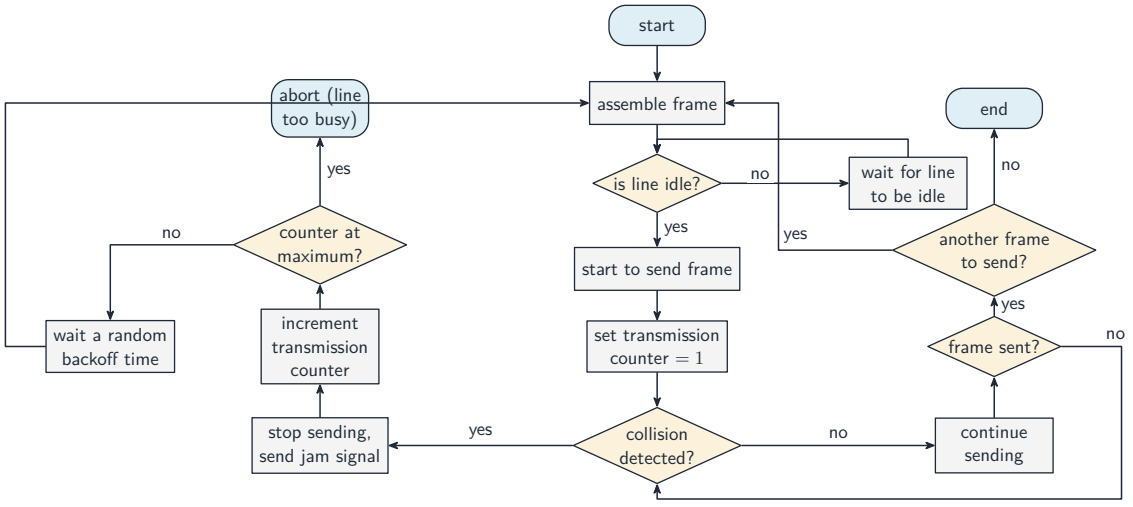

*A router connects a LAN to the internet or another network*

## Ethernet and CSMA/CD

Ethernet is the main wired LAN technology. On shared media a **collision** 冲突 can happen when two devices send at once. The protocol is **CSMA/CD** 载波侦听多路访问/冲突检测 (Carrier Sense Multiple Access with Collision Detection):

1. **carrier sense** — listen before sending; wait if the cable is busy.
2. **multiple access** — many devices share the medium.
3. **collision detection** — keep listening while sending; a clash is a collision.
4. on a collision, both stop, send a brief "jam" signal, then wait a **random backoff** time before retrying.

Modern switched Ethernet uses **full-duplex** 全双工 point-to-point links, so collisions no longer happen.




*The CSMA/CD process for handling collisions on shared media*

# Bit streaming

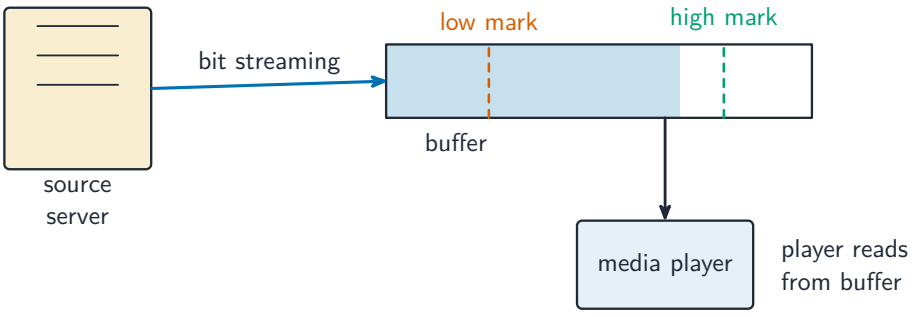
**Bit streaming** 流式传输 sends multimedia as a continuous stream that the receiver plays as it arrives, instead of downloading the whole file first.

- **real-time** (live): captured and streamed as it happens (live sport, video calls). You cannot rewind; low latency is vital.
- **on-demand**: pre-recorded on a server (YouTube, Netflix). You can pause and rewind; the server can **buffer** 缓冲 ahead.

Real-time streaming works as a short pipeline:

1. **capture and sample** the source (a camera or microphone).
2. **encode** it, using **compression** 压缩 to shrink the data.
3. **send** it across the network as packets.
4. the receiver buffers a little, then plays it live — **dropping any packet that arrives late**, because a live stream cannot wait for it.

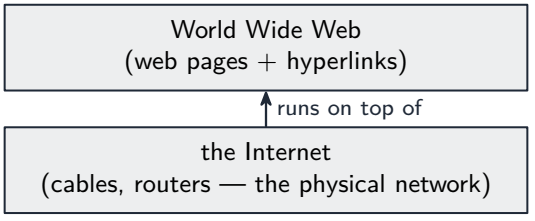
**Lossy** 有损 compression is used here: moving pictures hide small losses, and the stream must be small enough to fit the bandwidth.




*Data streams from the server into a buffer before the media player reads it*

# The internet and the World Wide Web

The **internet** 互联网 is a global **network of networks** using a common **protocol** 协议 suite (TCP/IP). The **World Wide Web** 万维网 (WWW) is a **service** that runs over it: hyperlinked documents identified by URLs, viewed in browsers via HTTP/HTTPS. Email and file transfer are other internet services that are not part of the WWW.




*The Web is one service running on top of the Internet*

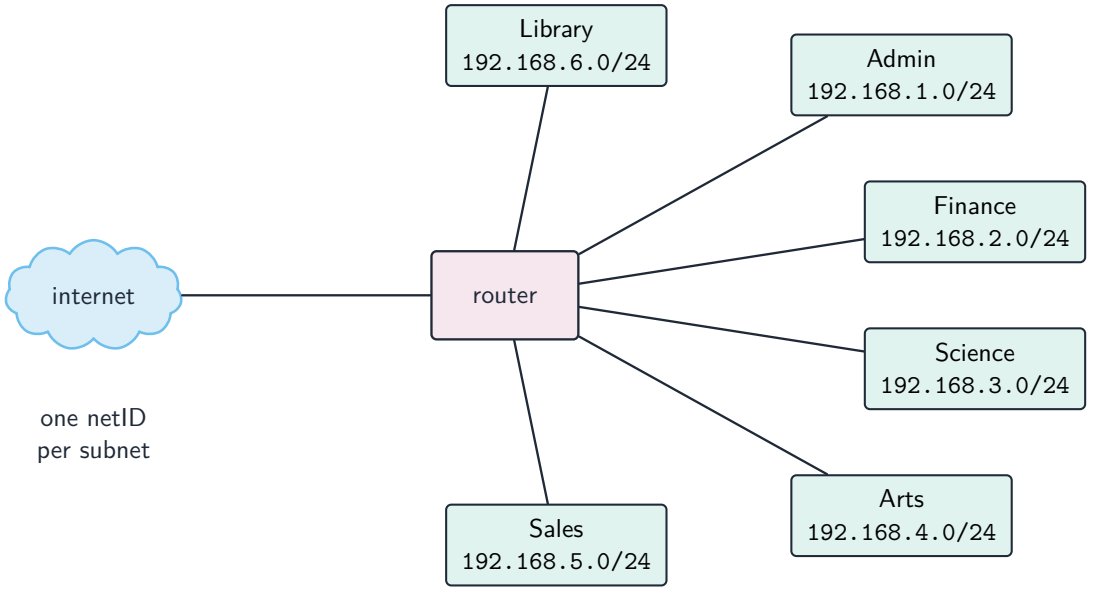
# IP addresses

An IP address uniquely identifies a device.

- **IPv4** — 32-bit, four denary numbers 0–255 (`192.168.1.10`); about $4.3 \times 10^9$ addresses (now exhausted).
- **IPv6** — 128-bit, eight groups of four hex digits; about $3.4 \times 10^{38}$ addresses.

## Subnetting

A network can be split into **subnets** 子网. The IP address splits into a network part and a host part, given by a **subnet mask** 子网掩码 (e.g. `255.255.255.0` = first 24 bits are network). Subnetting improves management, cuts broadcast traffic, and improves security.




*Splitting a network into subnets, one netID per department*

## Public vs private addresses

- **private** addresses are used within a LAN and are not routable on the internet (e.g. `192.168.0.0/16`).
- a **public IP address** is globally unique and routable, assigned by an **ISP** 互联网服务提供商.

Devices behind NAT with private addresses are not directly reachable from the internet, giving some protection.

### Static vs dynamic

- a **static IP address** is fixed; used for servers that must be found at a known address.
- a **dynamic IP address** is assigned by DHCP and may change; easier for client devices and uses a limited address pool efficiently.

**Worked example.** A host has IP address `192.168.10.130` with subnet mask `255.255.255.192`. Which network is it on, and is `192.168.10.200` on the same one? The mask's last octet, `192`, is `11000000` in binary, so the first **26** bits are the network part and the last 6 bits address the host. That makes the subnets step in blocks of $256 - 192 = 64$: `.0`, `.64`, `.128`, `.192`. The address `130` falls in the block starting at `.128`, so the host is on network **`192.168.10.128/26`**, whose usable hosts run `.129` to `.190` (`.191` is the broadcast address). `200` falls in the **next** block (`.192`), so it is on a **different** subnet and traffic between the two must pass through a **router**. Get the block size from the mask first (256 minus the mask octet) - guessing from the first three octets is what makes these go wrong.

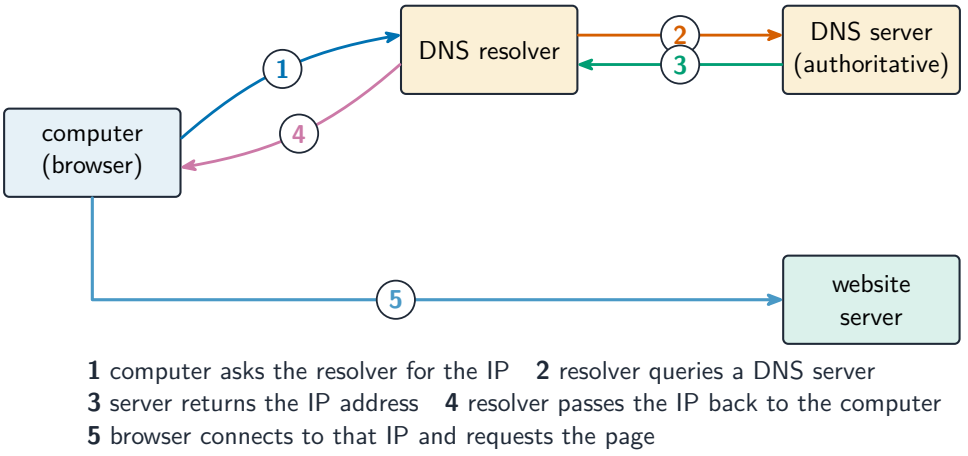
## URL and DNS

A **URL** 统一资源定位符 (Uniform Resource Locator) locates a resource on the WWW:

```
https://www.example.com/about/contact.html
protocol     domain name        path
```

- **protocol**: `http`, `https`, etc.
- **domain name** 域名: a readable server address.
- **path**: the resource on that server.

The **Domain Name System** 域名系统 (DNS, also called the Domain Name Service) is a distributed set of servers that turns **domain names into IP addresses**. When you type a URL, the browser asks a DNS resolver for the IP, which queries DNS servers (root → top-level → authoritative) until it finds it; the browser then connects to that IP and requests the path. DNS saves humans from memorising IP addresses and lets a site change server without changing its name.




*How DNS finds a website's IP address before the browser connects*

# Exam tips

- Distinguish **LAN vs WAN** and **client-server vs peer-to-peer** by who stores and controls the resources.
- Match each **topology** (bus, star, mesh) to its advantages and drawbacks (cost, reliability, collisions).
- Know the job of each device: a **switch** directs within a LAN by MAC address, a **router** routes between networks by IP.
- Explain **bit streaming** and why buffering is needed (data arrives at a different rate from playback).
- Distinguish **IPv4 vs IPv6** and public vs private addresses; DNS turns a URL into an IP address.

# Hardware

A-Level Computer Science

## Computers and their components

A general-purpose computer has four building blocks:

- **input devices** 输入设备 — get data in (keyboard, mouse, microphone, scanner, sensors).
- **output devices** 输出设备 — give results out (monitor, speakers, printer, actuators).
- **primary memory** 主存储器 — fast memory the **processor** 处理器 (CPU) reaches directly (RAM and ROM). Holds the running program and its data.
- **secondary storage** 辅助存储器 — slower, larger, keeps programs and data when not in use (hard disk, SSD, optical disc, USB stick).

*A keyboard: a common input device for typing text and commands*


*A mouse: a pointing input device*


*A flatbed scanner: an input device that turns a paper page into a digital image*

*A monitor: a common output device that displays the screen image*

## Embedded systems

An **embedded system** 嵌入式系统 is a computer built **into another device** to do one fixed job (washing machine, microwave, car engine unit, thermostat).

- **benefits**: optimised for one task (low power, small, cheap); reliable; fast to start; cheap in volume.
- **drawbacks**: limited to its one task; hard to update (its **firmware** 固件 may need special tools); often not repairable; sometimes weak security.

# Principal hardware devices

## Laser printer

A **laser printer** 激光打印机 scans the page image onto a charged photosensitive **drum** 感光鼓. **Toner** 墨粉 sticks to the charged areas, transfers to the paper, and is melted on by a fuser. Fast, sharp, high-volume.


*A laser printer: fast, sharp printing using a charged drum and toner*

## 3D printer

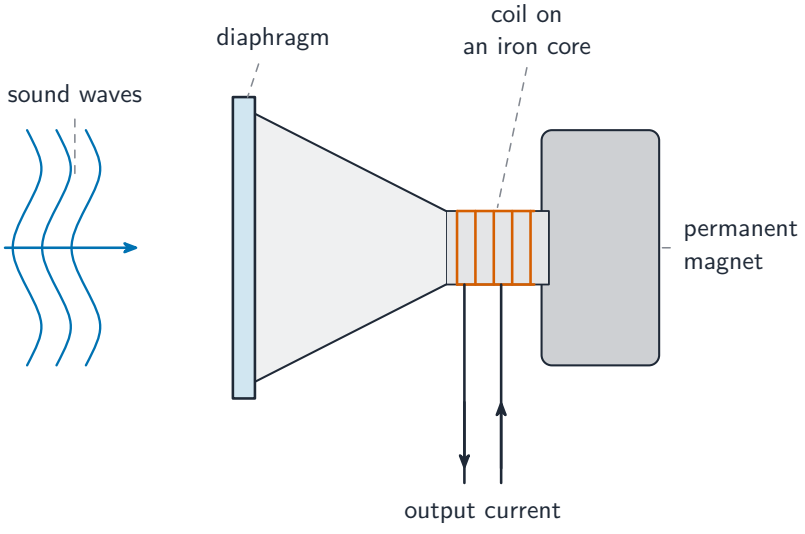
A **3D printer** 3D 打印机 builds an object **layer by layer**: FDM melts plastic filament through a nozzle; stereolithography cures liquid resin with a UV laser. Used for prototypes and custom medical parts.


*An FDM 3D printer builds an object layer by layer by melting plastic filament*

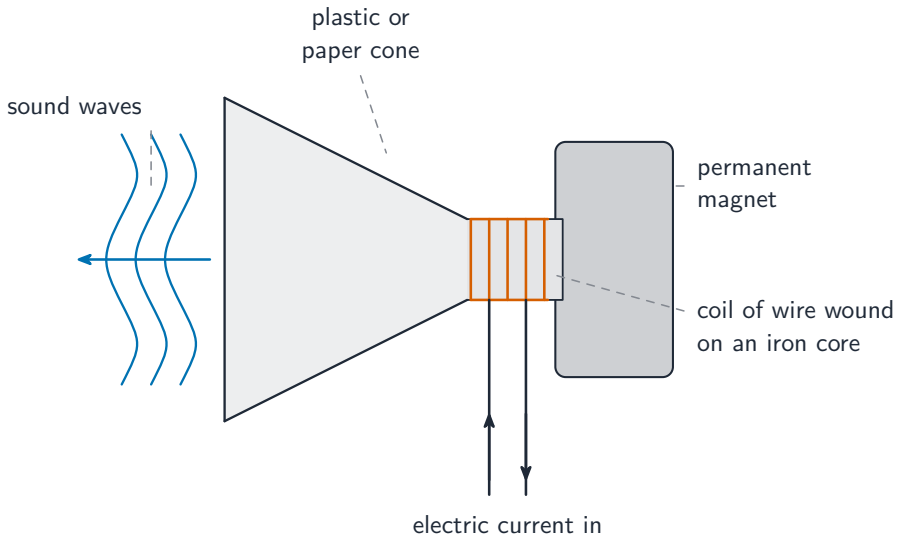
## Microphone and speakers

A **microphone** 麦克风 turns **sound into an electrical signal** (a diaphragm vibrates, changing **capacitor** 电容器 charge or coil position); the signal is digitised by an **analogue-to-digital converter** 模数转换器 (ADC). A speaker does the reverse — a varying signal drives a coil in a magnetic field, moving a cone to make sound.

*A microphone turns sound into an electrical signal*




*Inside a microphone: sound vibrates the diaphragm and coil to produce a current*

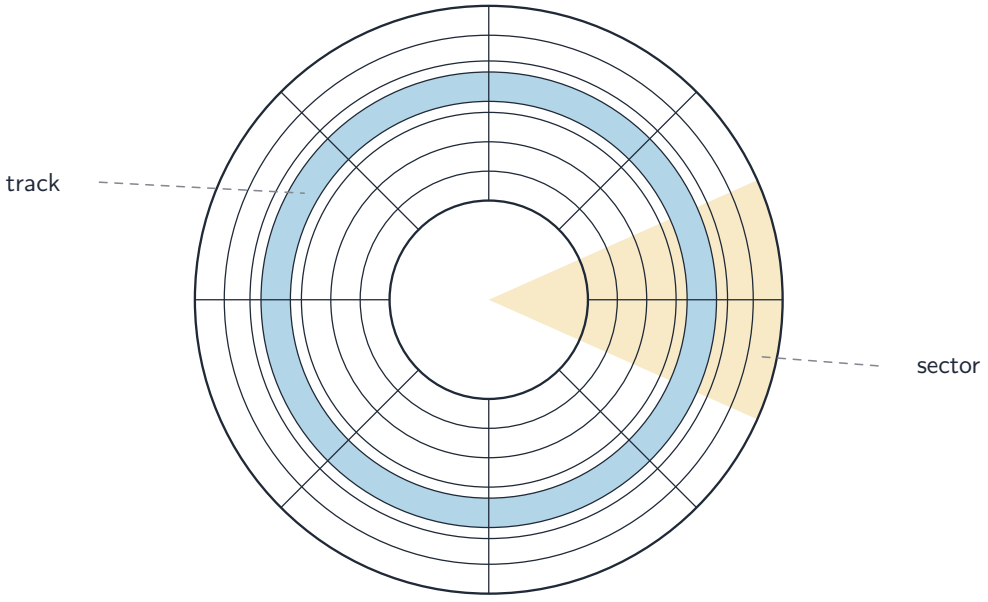



*Inside a loudspeaker: a varying current in the coil moves the cone to make sound*

## Magnetic hard disk (HDD)

A **hard disk** 硬盘 stores data on spinning platters coated with magnetic material. Each platter has **tracks** 磁道 divided into **sectors** 扇区. A **read/write head** 读写头 floats just above and magnetises tiny regions (write) or senses them (read). Cheap per gigabyte, but slower than SSDs and has moving parts.

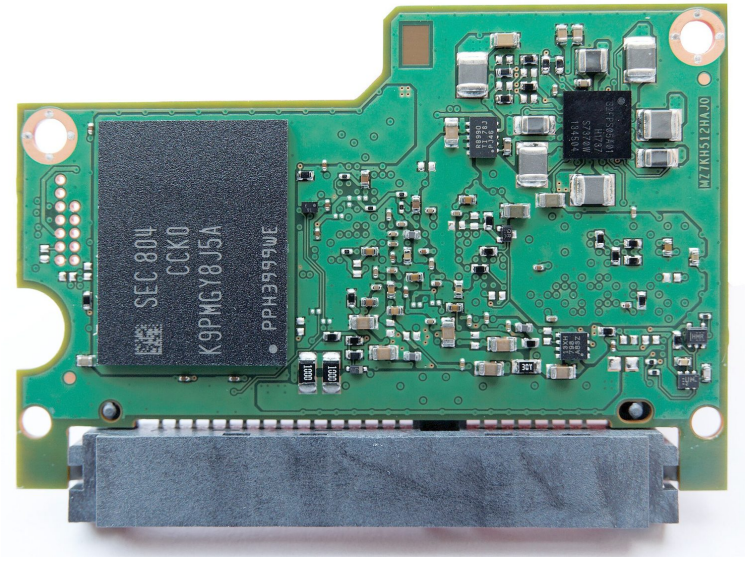
*An opened hard disk: the actuator arm carries the read/write head over a platter*




*Tracks and sectors on a hard disk platter*

## Solid-state (flash) memory

A **solid-state drive** 固态硬盘 stores data as charge in **transistors** 晶体管, with no moving parts. Faster random access than HDDs, tougher, lower power, but dearer per gigabyte; each cell wears out after many writes.




*Inside an SSD: data is stored in flash memory chips, with no moving parts (compare the hard disk above)*

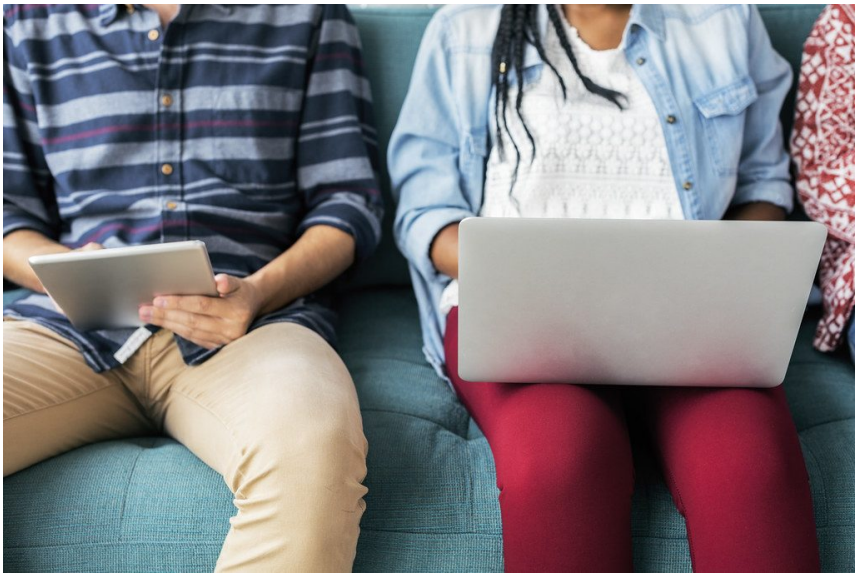
## Optical disc

A laser detects reflections from tiny pits on an **optical disc** 光盘 (CD, DVD, Blu-ray). The drive is an **optical disc reader/writer**: writing uses a stronger laser to change the surface's reflectivity.


*An optical disc drive: a laser reads tiny pits on a CD, DVD or Blu-ray disc*

## Touchscreen

A **touchscreen** 触摸屏 senses contact. **Resistive** 电阻式: two conductive layers pressed together; works with anything but is less accurate. **Capacitive** 电容式: a finger disturbs a charge field; accurate, multi-touch, used in phones.



*A touchscreen senses where a finger touches the glass*

### Virtual reality headset

A **virtual reality** 虚拟现实 (VR) headset has two small displays (one per eye) and motion sensors (**accelerometer** 加速度计, **gyroscope** 陀螺仪) that track head movement so the scene shifts as you look around.

*A virtual reality headset: two small displays and motion sensors track the head*

## Buffers

A **buffer** 缓冲 is memory that **holds data temporarily** while it moves between devices of different speeds. Example: the CPU writes a document to a printer buffer quickly, then is free to do other work while the printer prints from the buffer at its own pace. Buffers stop the fast device waiting for the slow one (also used in streaming, the keyboard, and disk access).

# RAM and ROM

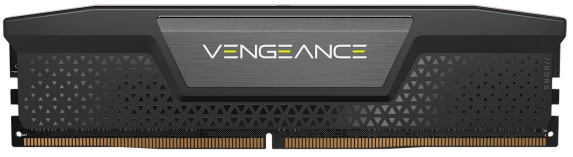
- **RAM** 随机存取存储器 (Random Access Memory) — **volatile** 易失性 (loses data without power). Holds the OS, running programs and their data; read and written constantly.
- **ROM** 只读存储器 (Read-Only Memory) — **non-volatile** 非易失性 (keeps data without power). Usually written once; holds firmware needed at start-up (the BIOS / boot loader).

**RAM**
volatile — lost at power off
read *and* write
holds running programs & data

**ROM**
non-volatile — kept at power off
read-only
holds start-up instructions

*RAM is volatile and read/write; ROM is non-volatile and read-only*

ROM starts the system; RAM then holds the active work.




*A RAM module (DIMM) plugs into the motherboard as the computer's fast main memory*

# SRAM vs DRAM

- **SRAM** 静态 RAM (Static RAM) stores each bit in a **flip-flop** 触发器 of several transistors. Fast, but expensive and not dense. Used for CPU **cache** 高速缓存.

- **DRAM** 动态 RAM (Dynamic RAM) stores each bit as charge on a tiny capacitor. Cheaper and denser but slower, and must be **refreshed** 刷新 (rewritten) thousands of times a second. Used for main memory.

Use SRAM for small fast memory (cache); DRAM for large main memory.

# PROM, EPROM and EEPROM

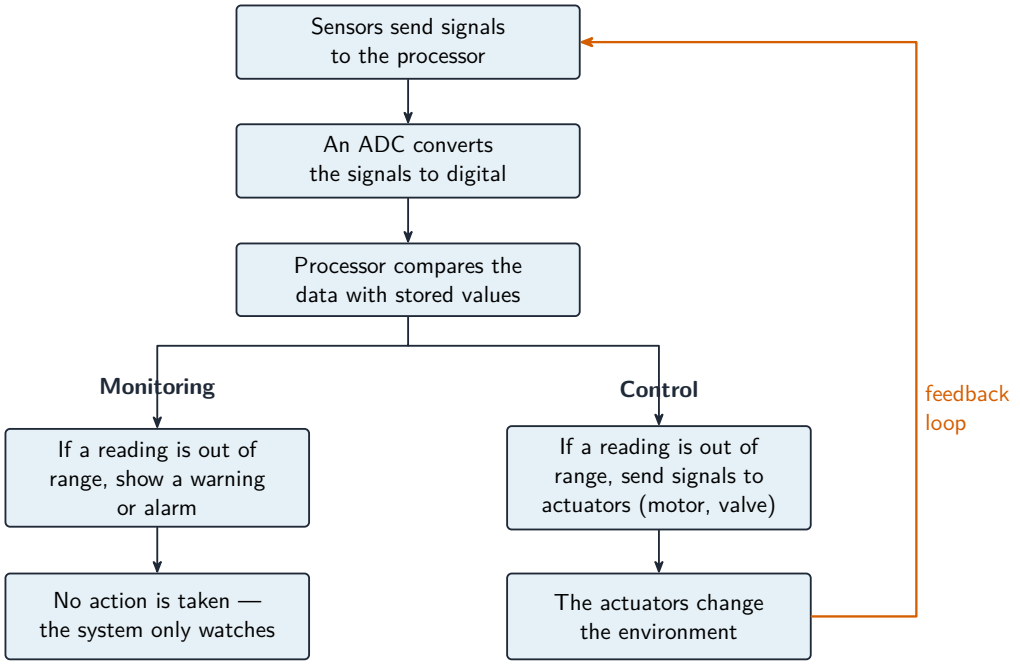
ROM variants you can program after manufacture:

- **PROM** (Programmable ROM) — written **once** (fuses burned by a programmer); cannot be changed.
- **EPROM** (Erasable Programmable ROM) — erased by **strong UV light** through a window, then rewritten (whole chip at once).
- **EEPROM** (Electrically Erasable Programmable ROM) — erased and rewritten **electrically**, a byte at a time, in circuit. Flash memory is a derivative optimised for block erase.

# Monitoring and control systems

Both read **sensors**; the difference is what they do next.

- **monitoring** 监控 — collects and reports data but **takes no action** (a weather station logging readings).
- **control system** 控制系统 — uses sensor data to **decide and act** through actuators, usually in a feedback loop (a thermostat turning a boiler on/off).

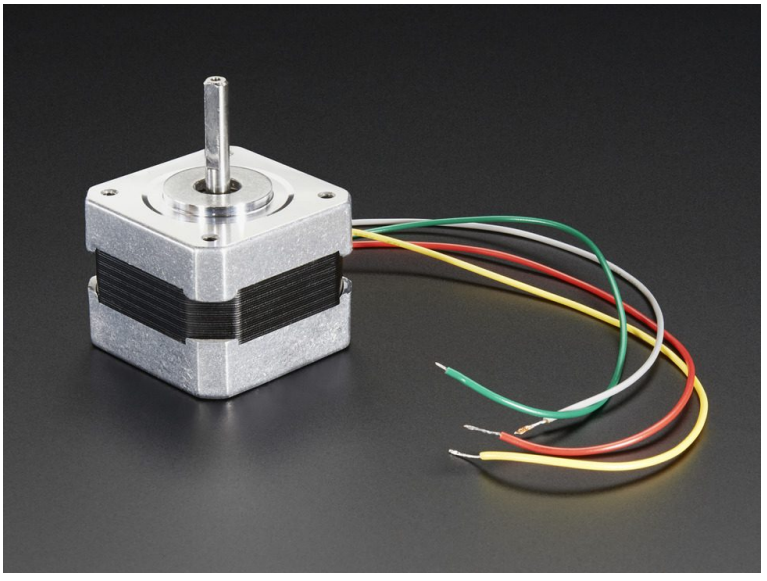



*Monitoring reports data; a control system acts through a feedback loop*

## Sensors and actuators

A **sensor** 传感器 turns a physical quantity into a signal: temperature (a **thermistor** 热敏电阻 or thermocouple), pressure (strain gauge), infra-red, sound. Analogue signals need an ADC first. An **actuator** 执行器 does the reverse — turns a signal into an action (a motor, valve, heater, buzzer).

*A thermistor: a temperature sensor whose resistance changes with heat*



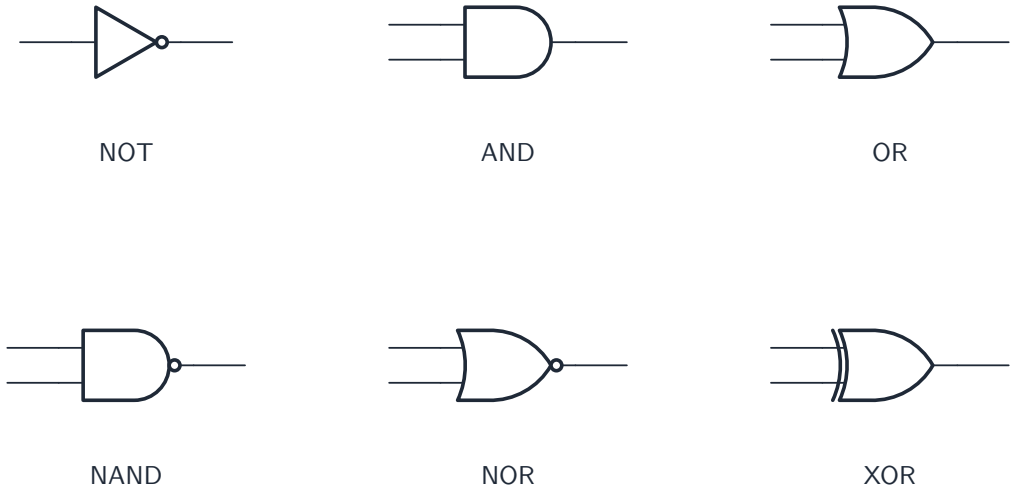
*A small electric motor: an actuator that turns a signal into movement*

### Feedback

In a control system the actuator changes the environment, which the sensors then re-measure — a **feedback** 反馈 loop. Without feedback the system cannot correct itself or know when to stop (a thermostat with no temperature feedback would heat forever).

## Logic gates

A **logic gate** 逻辑门 is a small circuit that does one **Boolean** 布尔 operation. Inputs and outputs are 0 (false, low) or 1 (true, high). Know the symbol, function and **truth table** 真值表 for each gate.




*The symbols for the six logic gates*

### NOT (inverter)

| A | NOT A |
|---|---|
| 0 | 1 |
| 1 | 0 |

### AND — output 1 only if all inputs are 1

| A | B | A AND B |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

### OR — output 1 if at least one input is 1

| A | B | A OR B |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

### NAND (NOT AND) — output 0 only when all inputs are 1

| A | B | A NAND B |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

### NOR (NOT OR) — output 1 only when all inputs are 0

| A | B | A NOR B |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 0 |

### XOR (Exclusive OR, also called EOR) — output 1 if the inputs are different

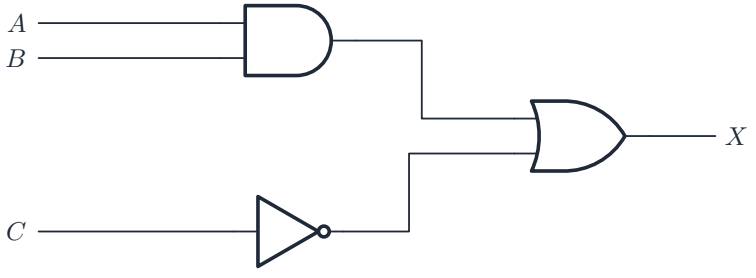
| A | B | A XOR B |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

## Logic circuits

A **logic circuit** 逻辑电路 is a network of gates that carries out a Boolean expression. You should be able to move between a problem statement, a logic expression, a truth table, and a circuit diagram.

### From expression to circuit

Draw one gate per operator and wire them up. For $X = (A \text{ AND } B) \text{ OR } (\text{NOT } C)$: a NOT gate on $C$, an AND gate on $A$ and $B$, then an OR gate on the two results.




*Gates wired together to carry out a Boolean expression*

### From circuit to expression

Work forwards from the inputs, labelling each gate's output, until you reach the final output.

### From circuit to truth table

For $n$ inputs there are $2^n$ rows. List every input combination; for each, work out the internal gates then the output.

### From truth table to expression (sum of products)

For each row that outputs 1, write an AND of the inputs (with NOT on any input that is 0 in that row); OR these together. Example: a table that is 1 only on $(A = 0, B = 1)$ and $(A = 1, B = 0)$ gives $\overline{A}B + A\overline{B}$, which is $A$ XOR $B$.

### From a problem statement

Turn the English into a Boolean expression first: "A and B" → A AND B; "A or B or both" → A OR B; "exactly one of A and B" → A XOR B; "neither A nor B" → A NOR B; "not both" → A NAND B.

**Worked example.** A machine's alarm $X$ sounds when the guard is open ($A = 1$) **and** either the motor is running ($B = 1$) or the temperature is high ($C = 1$). Write the Boolean expression, and give the rows where $X = 1$. Turn the English into logic one clause at a time: "either B or C" is $B+C$, and "A **and** that" is $X = A \cdot (B+C)$. For the rows, $X = 1$ needs $A = 1$ **and** at least one of $B$, $C$ equal to 1 - so $(A, B, C) = (1, 0, 1)$, $(1, 1, 0)$ and $(1, 1, 1)$, three rows out of eight. Notice $A = 0$ can never sound the alarm, whatever $B$ and $C$ do. **Bracket the OR** before ANDing it: $X = A \cdot B + C$ is a different circuit altogether, one that would sound the alarm on a high temperature even with the guard **closed**.

## Exam tips

- Distinguish **RAM (volatile, read/write)** from **ROM (non-volatile, holds the bootstrap)**; SRAM (cache, faster) from DRAM (main memory, needs refreshing).
- For a **logic circuit**, build the Boolean expression gate by gate, then a truth table covering **every** input combination.
- Learn the symbol, expression and truth table for each gate (AND, OR, NOT, NAND, NOR, XOR).
- Explain a **buffer** (a temporary store bridging two different speeds) and the role of an interrupt.

# Processor Fundamentals

A-Level Computer Science

## Von Neumann architecture

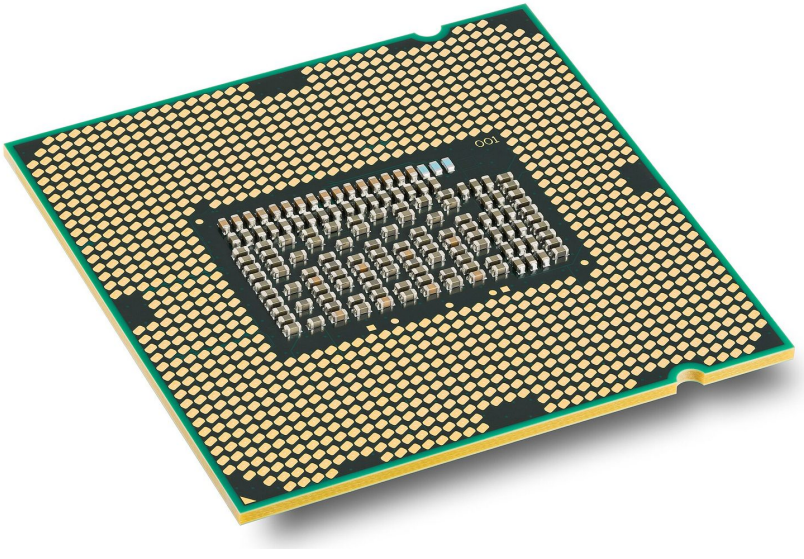
The **Von Neumann architecture** 冯 · 诺依曼体系结构 underlies almost every general-purpose computer:

- a **single memory** — the **Immediate Access Store** 立即存取存储器 (IAS) — holds both **program instructions** and **data** (the **stored program** 存储程序 concept).
- a **processor** 处理器 (CPU) fetches instructions from memory and runs them one at a time.
- instructions run **in order** unless a branch changes the flow.

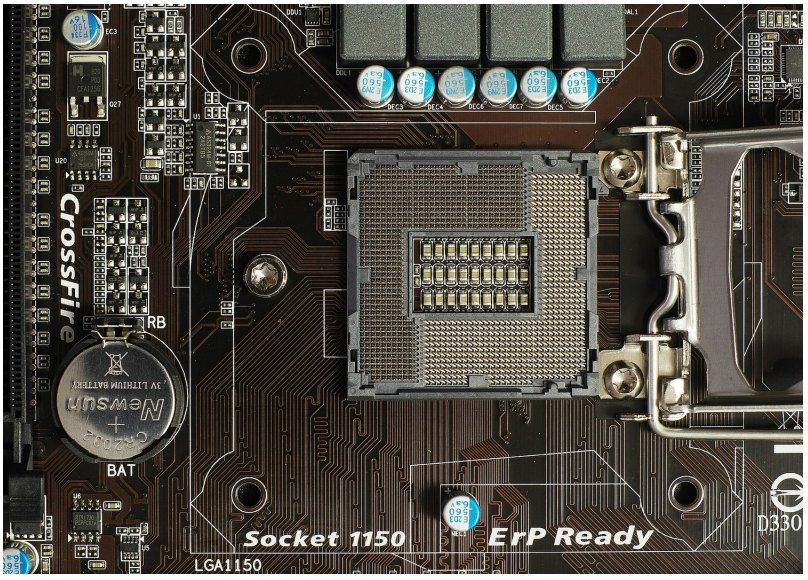
The stored-program idea is what makes a computer flexible: change the program and you change what it does, with no rewiring.

## The CPU's main parts

All of these parts sit inside one small chip. The diagram later in this section shows how they connect; the photo below shows the real thing.



*A modern CPU: the whole processor is one small chip (here seen from below, showing the contacts)*




*The matching CPU socket on the motherboard: the chip's contacts press onto these pins*

## Arithmetic and Logic Unit (ALU)

The **ALU** 算术逻辑单元 does the arithmetic (add, subtract, …) and logic (AND, OR, comparisons). It takes operands from **registers** 寄存器 and puts results back in a register.

## Control Unit (CU)

The **control unit** 控制单元 **decodes** each instruction and sends the **control signals** to carry it out — opening data paths, telling the ALU what to do, and controlling memory reads and writes.

## System clock

The clock sends a steady stream of pulses that keep the CPU in step. Each instruction takes a fixed number of cycles, and the **clock speed** 时钟频率 (e.g. 3.8 GHz) is one factor in performance.

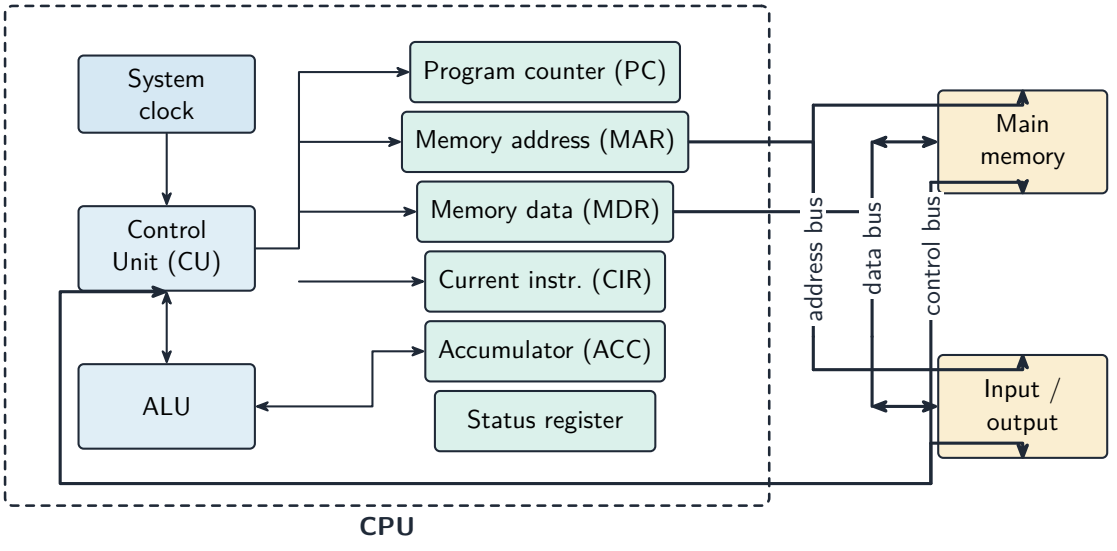
## Registers

Registers are tiny, very fast stores inside the CPU. The **special purpose registers** 专用寄存器 each have a fixed job in the cycle:

- **Program Counter** 程序计数器 (PC) — the address of the **next** instruction.

- **Memory Address Register** 内存地址寄存器 (MAR) — the address being read or written.
- **Memory Data Register** 内存数据寄存器 (MDR) — the data going to or from memory.
- **Current Instruction Register** 当前指令寄存器 (CIR) — the instruction being decoded.
- **Accumulator** 累加器 (ACC) — the value the ALU is working on.
- **Status Register** 状态寄存器 — holds **flags** 标志 (carry, zero, negative, overflow) used by branches.
- **Index Register** 变址寄存器 — an offset used in indexed addressing.

**General-purpose registers** 通用寄存器 are used by the programmer for temporary values during a calculation. Movements of data between registers and memory are written in **register transfer** 寄存器传送 notation — e.g. `MAR ← [PC]` ("copy the contents of PC into MAR").



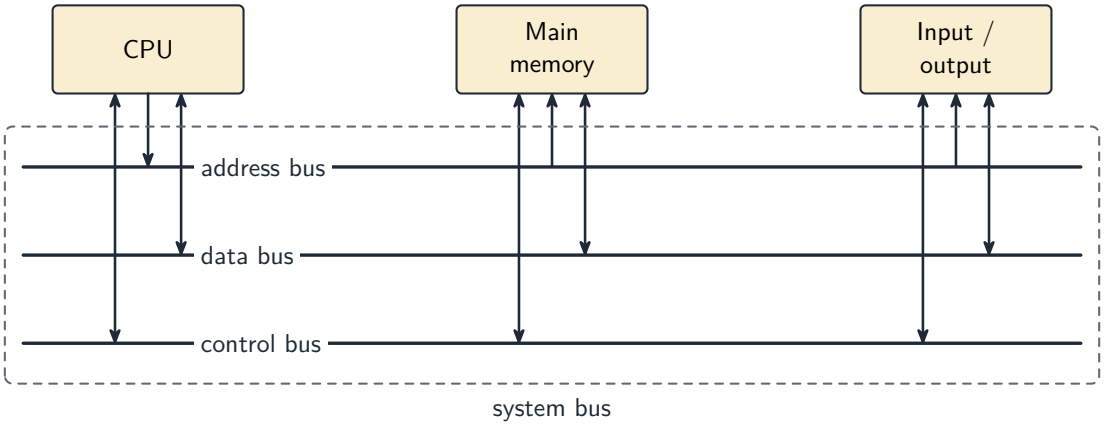

*The Von Neumann CPU: registers, control unit and ALU linked by buses*

# Buses

Three internal **buses** 总线 (sets of parallel wires) connect the parts:

- **address bus** 地址总线 — carries the memory address. **One-way** (CPU → memory).
- **data bus** 数据总线 — carries the data. **Two-way**.
- **control bus** 控制总线 — carries control signals (read, write, interrupt). **Two-way**.

An $n$-bit address bus can reach $2^n$ memory locations. The data-bus width sets how many bits move per access (often the word size).




*The three system buses connecting the CPU, memory and input/output*

# What affects performance

- **clock speed** — more cycles per second.
- **number of cores** 核心 — a multi-core CPU runs several threads at once.
- **word size** 字长 — a 64-bit CPU handles 64-bit chunks per cycle and can address far more memory than a 32-bit one.
- **amount of RAM** 随机存取存储器 — more RAM holds more of the working set; too little forces the OS to **page** 页 to disk.
- **cache memory** 高速缓存 size — more cache cuts average memory access time.
- **secondary storage** 辅助存储器 type — an SSD loads programs far faster than an HDD.
- **bus width and speed** — wider/faster buses move data more quickly.

Match the specs to the workload: a quad-core beats a dual-core on parallel work, but

higher per-core speed wins on single-threaded work.

## Ports

A **port** 端口 is a physical socket for connecting a **peripheral** 外围设备:

- **USB** (Universal Serial Bus) — general-purpose (keyboards, drives, phones).
- **HDMI** (High Definition Multimedia Interface) — digital video and audio to a screen.
- **VGA** (Video Graphics Array) — older analogue video output to a monitor.
- **Ethernet (RJ-45)** — wired LAN. Audio jacks — headphones/microphone.

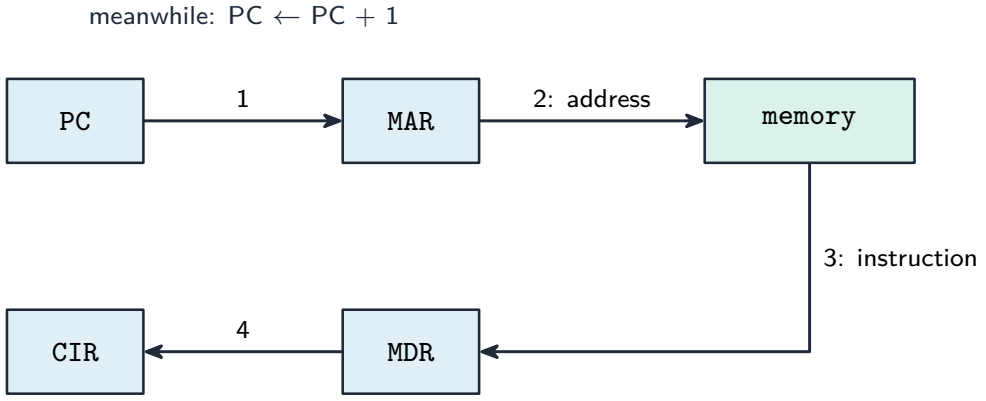
Different ports use different signals, so an HDMI cable will not fit a USB socket. USB-C is unusual in carrying video, data and power.

## Fetch-Execute cycle

The CPU repeats the **fetch-execute cycle** 取指-执行周期, one run per machine instruction.

## Fetch

1. the PC's address is copied to the MAR.
2. the PC is **incremented** to point to the next instruction.
3. a **read** signal goes over the control bus.
4. memory puts the instruction on the data bus.
5. it is copied into the MDR, then into the CIR.



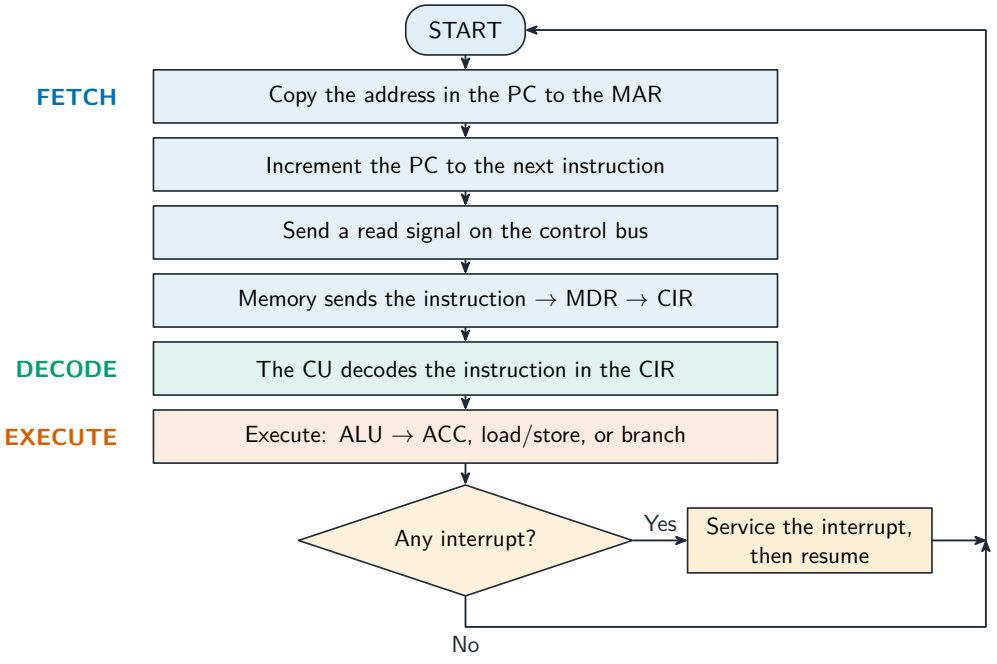

*The register transfers in a fetch: PC → MAR → memory → MDR → CIR, with the PC incremented*

## Decode

The CU decodes the instruction in the CIR — what operation, and which operands or addresses.

### Execute

The CU carries it out: arithmetic/logic goes to the ALU (result to the ACC); a load/store moves data between memory and a register; a branch changes the PC. Then the cycle repeats.




*The fetch-execute cycle, with a check for interrupts each time*

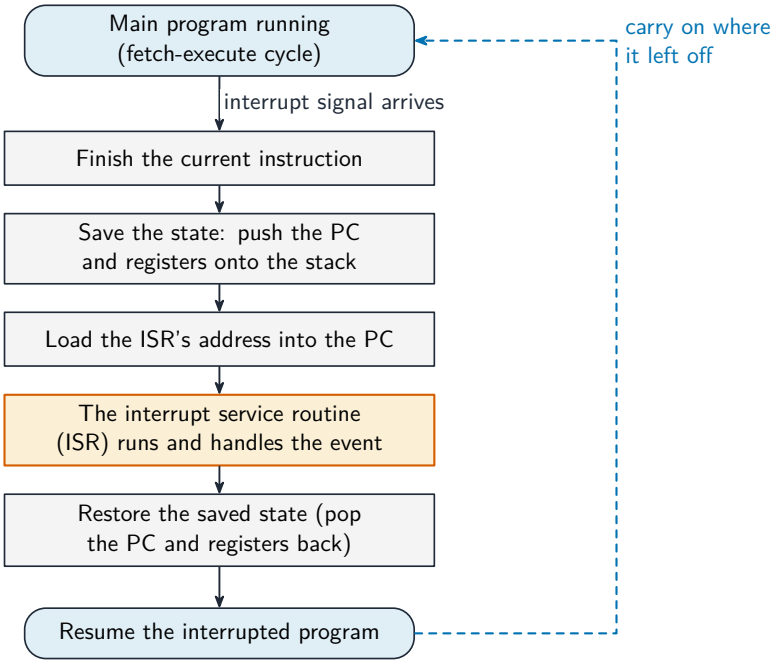
## Interrupts

An **interrupt** 中断 is a signal that **pauses** the normal cycle so the CPU can handle an urgent event (a key press, a packet arriving, a hardware fault, division by zero, the OS timer).

Handling one:

1. finish the current instruction.
2. **save the state** (PC and registers).
3. load the address of the **interrupt service routine** 中断服务程序 (ISR) into the PC and run it.
4. the ISR handles the event.
5. **restore** the saved state and carry on.

Interrupts let the system respond promptly without the CPU constantly checking devices, and are how the OS multitasks.




*How an interrupt fits into the fetch-execute cycle*

# Assembly language and machine code

The CPU actually runs **machine code** 机器码 — bit patterns, specific to one architecture. **Assembly language** 汇编语言 is a readable form, with one instruction per machine instruction, written using **mnemonics** 助记符 like `LDD`, `ADD`, `JMP`. An **assembler** 汇编器 translates it to machine code.


*An assembler turns mnemonics into machine-code bit patterns*

## Two-pass assembler

A two-pass assembler reads the source twice:

- **pass 1** builds a **symbol table** 符号表: each time a **label** 标签 (like `LOOP:`) appears, record its address; no code yet.
- **pass 2** generates code: translate each instruction, and when one refers to a label (like `JMP LOOP`), look up its address in the symbol table.

Two passes handle **forward references** 前向引用 (a jump to a label defined later).

### Example instruction set

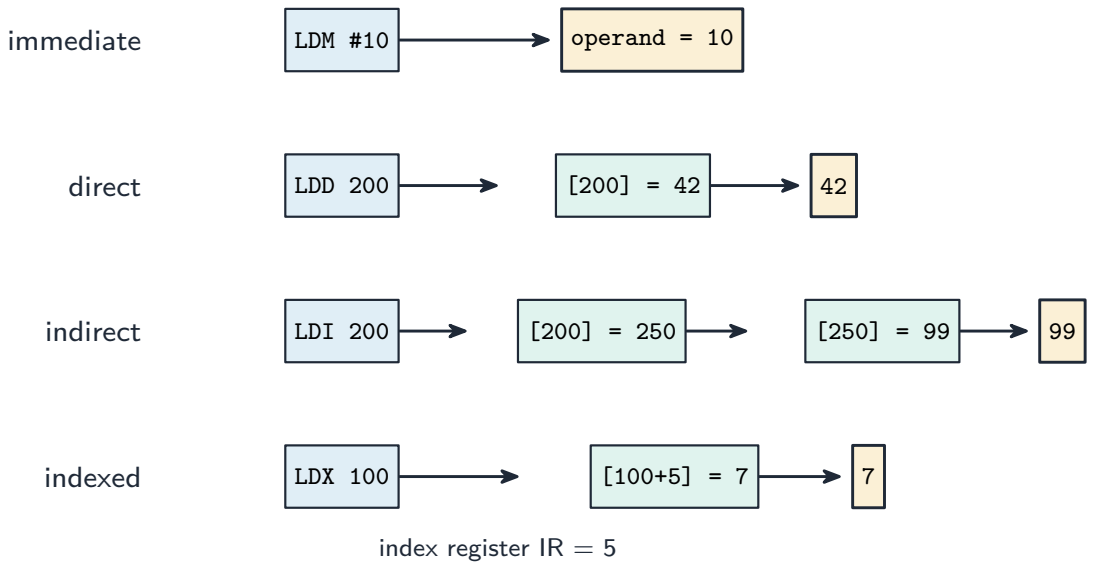
Cambridge uses a small generic set: data movement (`LDD`, `LDM`, `LDI`, `LDX`, `STO`, `MOV`), arithmetic (`ADD`, `SUB`, `INC`, `DEC`), logic/bit (`AND`, `OR`, `XOR`, `LSL`, `LSR`), compare and branch (`CMP`, `JMP`, `JPE`, `JPN`), I/O (`IN`, `OUT`), and `END`. The exact mnemonics are given in the paper's reference table.

## Addressing modes

The **addressing mode** 寻址方式 (the **modes of addressing**) says how the CPU finds the operand:

- **immediate addressing** 立即寻址 — the operand is the value in the instruction. `LDM #10` loads 10.
- **direct addressing** 直接寻址 — the instruction holds an address; the operand is the value there. `LDD 200`.
- **indirect addressing** 间接寻址 — the instruction holds an address that holds **another** address, which is the data. `LDI 200`.
- **indexed addressing** 变址寻址 — effective address is `address + index register`; used for arrays. `LDX 100` with IR = 5 reads address 105.

(**Relative addressing** 相对寻址 gives the address as an offset from the PC — used for jumps.)




*How each addressing mode reaches its operand — immediate, direct, indirect and indexed*

**Worked example.** Memory holds: location 200 = 250, location 250 = 99, location 105 = 7. The index register holds 5. What is in the accumulator after each of `LDM #200`, `LDD 200`, `LDI 200` and `LDX 100`? Follow how far each mode has to look. `LDM #200` is **immediate** - the operand *is* the number written in the instruction, so the accumulator holds **200**. `LDD 200` is **direct** - go to location 200 and take what is there: **250**. `LDI 200` is **indirect** - location 200 holds 250, which is *another address*, so go on to location 250: **99**. `LDX 100` is **indexed** - add the index register to the address, $100 + 5 = 105$, and read location 105: **7**. Count the hops to keep them apart: immediate 0, direct 1, indirect 2, indexed 1 (once the index has been added).

# Tracing an assembly program

To **trace** it: make a table with columns for the PC, ACC, index register, each variable and any flags. Step through the instructions, updating the table after each; follow branches when they change the PC; stop at `END`. A common pattern is a loop over an array using indexed addressing.

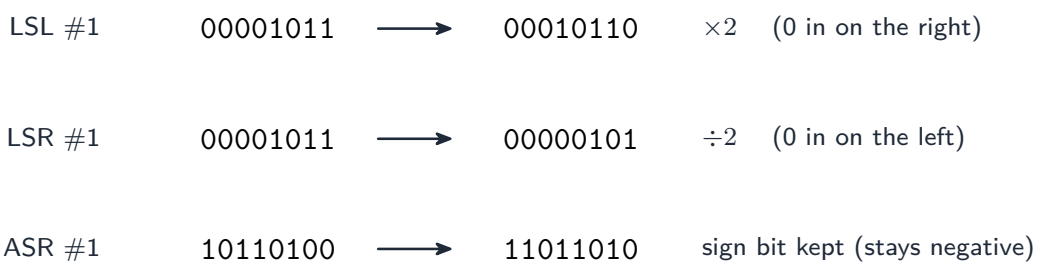
# Binary shifts

A **logical shift** 逻辑移位 moves all the bits left or right by some places, filling new positions with 0.

- **left shift by 1** (`LSL #1`) — bits move left, a 0 enters on the right; for an unsigned number this is $\times$ **2**.
- **right shift by 1** (`LSR #1`) — bits move right, a 0 enters on the left; for an unsigned number this is **integer $\div$ 2**.

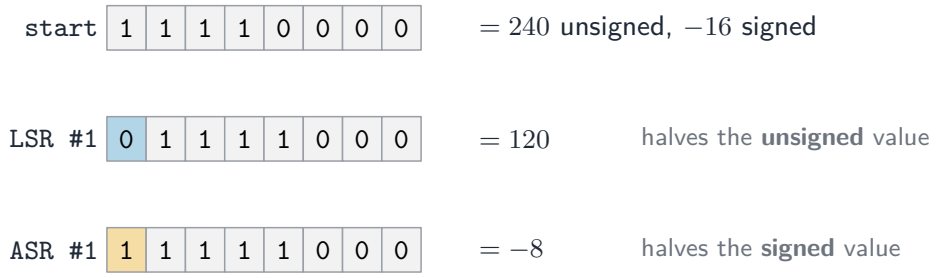
Shifting by $n$ places multiplies or divides by $2^n$. Example: `00001011` (11) `LSL #1` $\rightarrow$ `00010110` (22).

An **arithmetic right shift** keeps the sign bit so a negative signed number stays negative. A **cyclic shift** 循环移位 (rotate) feeds the bit that drops off one end back in at the other end, so no bits are lost.




*Logical left ($\times 2$), logical right ($\div 2$) and arithmetic right (keeps the sign bit)*

The difference between the two right shifts is a single bit. Take `11110000`, which is 240 read as unsigned and $-16$ read as signed. `LSR #1` brings in a 0 and gives `01111000` = 120, which is the correct half of 240. `ASR #1` copies the sign bit instead and gives `11111000` = $-8$, which is the correct half of $-16$. Neither is wrong — each halves the value under one reading.



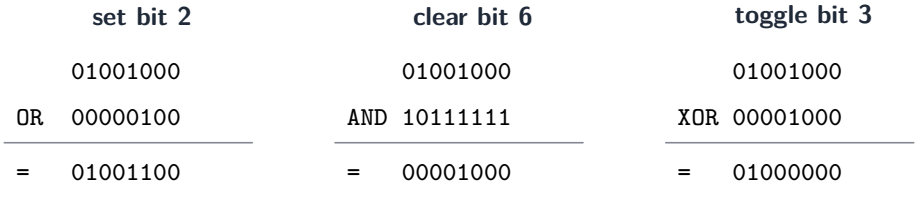

The two results differ in **one bit** — the one that entered on the left. **LSR** always brings in a 0; **ASR** copies the sign bit, so a negative number stays negative and $-16 \div 2$ comes out right.

*Logical and arithmetic right shift on the same byte: only the bit that enters on the left differs*

## Bit manipulation for monitoring/control

Embedded devices often use one **bit** 位 of a register per signal (e.g. bit $n$ = LED $n$). Using a **mask** 掩码 — **bit masking** — you can:

- **set** bit $n$: `R = R OR` a mask with bit $n$ set.
- **clear** bit $n$: `R = R AND` a mask with bit $n$ clear and the rest set.
- **toggle** bit $n$: `R = R XOR` a mask with bit $n$ set.
- **test** bit $n$: `R AND` the mask, then check if the result is non-zero.




*Set a bit with OR, clear it with AND, toggle it with XOR — each using a mask*

Bit manipulation is fast, uses little memory, and lets one byte hold up to 8 on/off states.

# Exam tips

- Learn the **fetch-execute cycle** in register-transfer terms (PC, MAR, MDR, CIR, ACC) and what increments the PC.

- Name each register's job; the **address bus is one-way**, the data bus is two-way.
- Distinguish the **addressing modes** (immediate, direct, indirect, indexed) — a frequent question.
- Explain how clock speed, number of cores, cache size and word length affect **performance**.
- For a **binary shift**, state whether it is logical or arithmetic; a left shift multiplies by 2, a right shift divides by 2.

# System Software

A-Level Computer Science

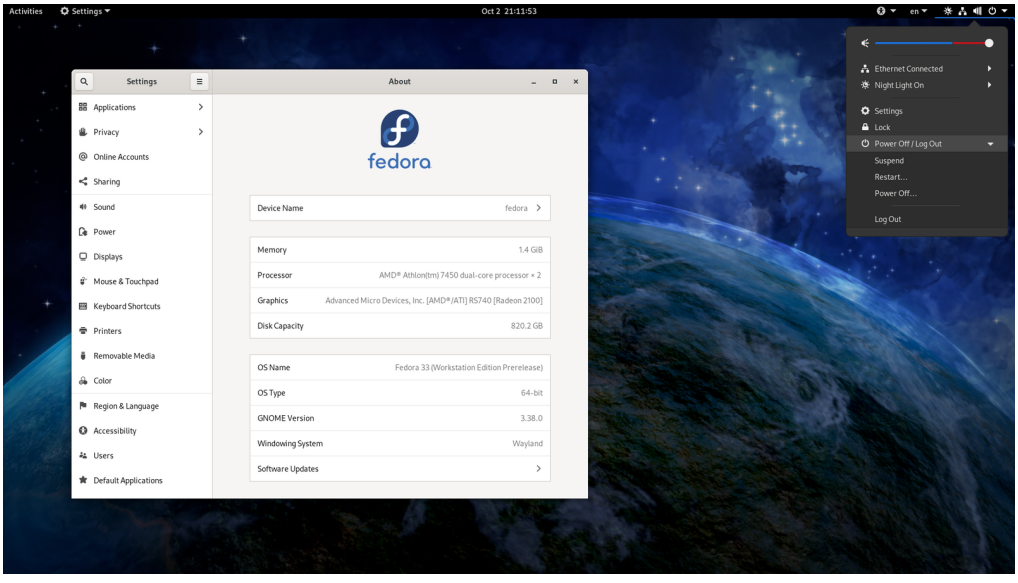
## Operating systems

### Why a computer needs an OS

Hardware on its own can only fetch and run instructions — it knows nothing about files, programs, networks or users. The **operating system** 操作系统 (OS) is the **software layer** that:

- **manages the hardware** (**processor** 处理器, memory, I/O, storage) for the running programs.
- **provides services** (file system, network, user accounts) through a clear interface, so programs need not talk to the hardware directly.
- **provides a user interface** (command line, GUI, touch).
- **lets several programs share the hardware safely** — each gets fair CPU time and is kept out of the others' memory.

Without an OS, every program would need its own drivers, and only one program could safely run at a time.

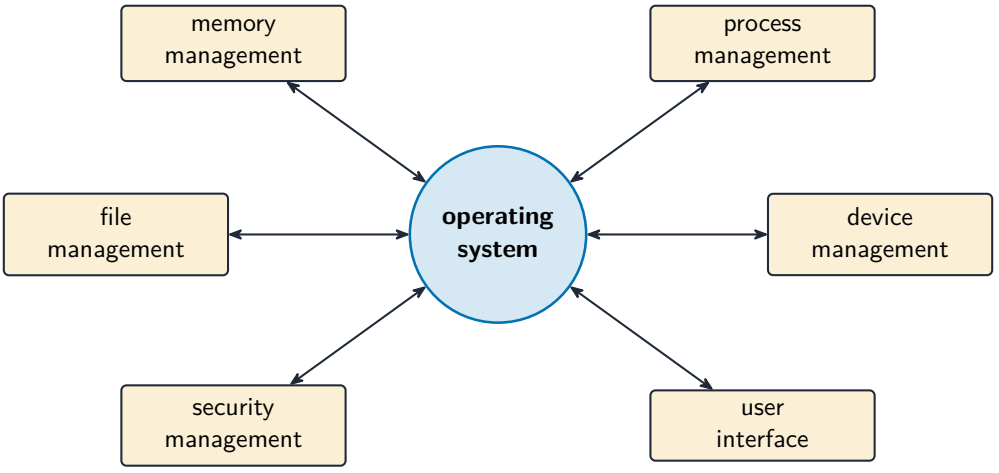



*A desktop operating system manages the screen, files and programs for the user*

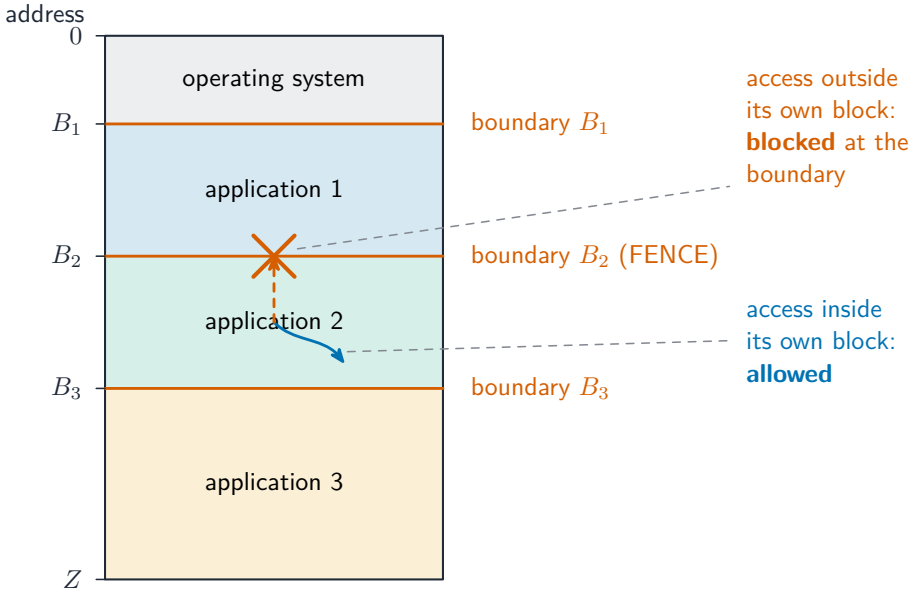

*A smartphone runs a mobile operating system such as Android*

## Key management tasks

- **process management** 进程管理 — load programs, schedule them on the CPU, switch between them, and kill misbehaving ones. (A running program is a **process** 进程.)
- **memory management** 内存管理 — give memory to processes, keep them apart, and use **virtual memory** 虚拟内存 / **paging** 分页 so the working set can exceed physical **RAM** 随机存取存储器.
- **file management** — organise files and folders on **secondary storage** 辅助存储器, control permissions, prevent write corruption.
- **device management** (hardware management) — handle I/O and peripherals through **device drivers** 设备驱动, buffer data, manage interrupts, and give a uniform interface.
- **security management** — accounts, permissions, firewall, encryption.
- **user interface** and **networking**.



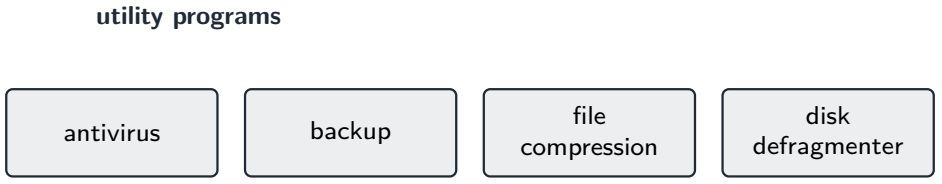

*The main jobs the operating system manages*




*Memory protection keeps each application in its own block of memory*

## Utility software

Most OSes include **utility programs** 实用程序 to maintain the system:



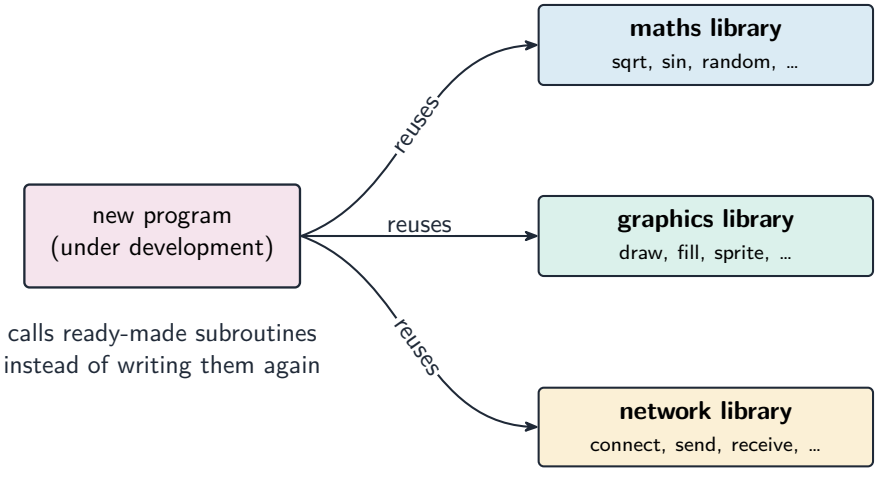

*Utility programs: antivirus, backup, compression and defragmenter*

- **disk formatter** — a **file / disk management** tool that prepares a disk (sets up its file system); related tools copy, move and delete files.
- **defragmentation software** (**disk defragmenter** 碎片整理) — moves the pieces of fragmented files together on a hard disk to cut seek time (not useful on SSDs).
- **disk contents analysis / disk repair software** — check disk integrity and fix file-system errors and bad sectors.
- **back-up software** (**backup** 备份) — copies user data elsewhere so it can be recovered.
- **virus checker** (**antivirus** 杀毒软件) — scans for malware and quarantines threats.
- **firewall** 防火墙 — filters network traffic by rules.
- **file compression** (**compression** 压缩) / archiving; **system monitor**; **updates**.

Bundling these with the OS saves the user installing each one.

# Program libraries

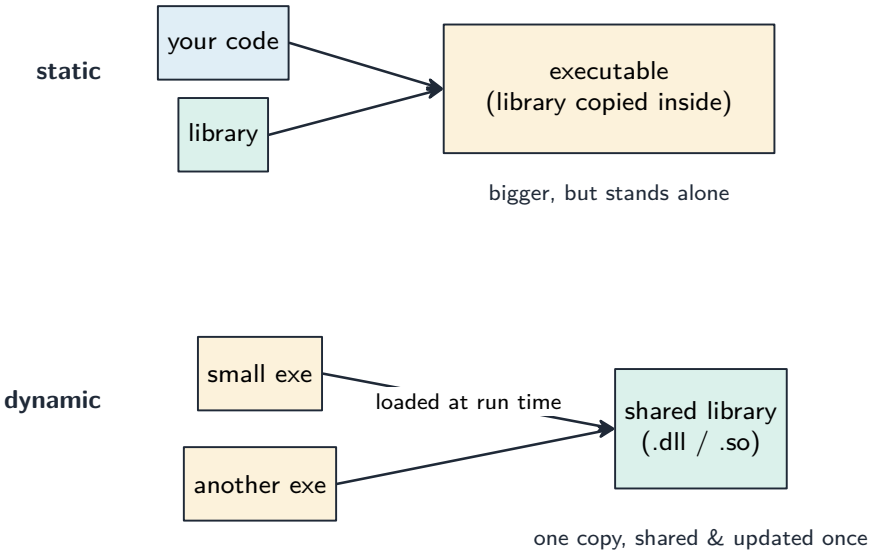
A **program library** 程序库 is pre-written code (**subroutines** 子程序, classes, modules) that programs reuse instead of writing it themselves — e.g. a maths library, a network library, a graphics library.




*A new program reusing ready-made routines from libraries*

Benefits: **saves time** (off-the-shelf code), **reliable** (well-tested, widely used), and **standardised** (consistent behaviour).

- a **static library** 静态库 is copied into the executable at compile time (stands alone, but larger and needs rebuilding to update).
- a **dynamic library** 动态库 (DLL, Dynamic Link Library; `.so`) is loaded at run time (smaller executables, shared by many programs, updated once for all).




*Static: the library is copied into the executable. Dynamic: a shared library file is loaded at run time*

# Language translators

You write source code; the computer runs **machine code** 机器码. A **translator** 翻译器 converts between them.

## Assembler

An **assembler** 汇编器 translates **assembly language** 汇编语言 into machine code, one instruction per instruction. Used for low-level code (embedded systems, drivers).

## Compiler

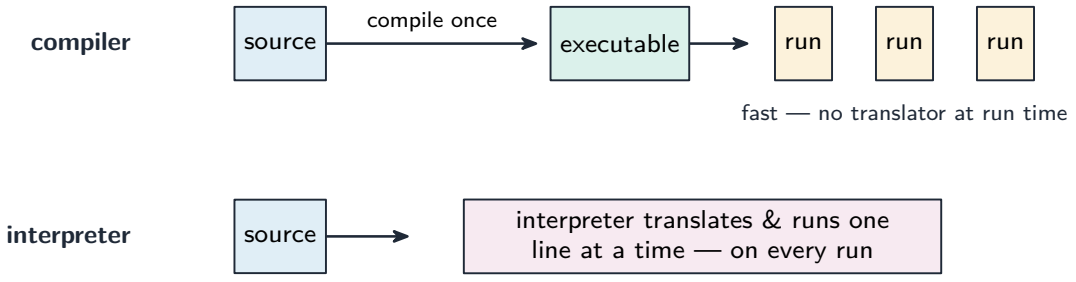
A **compiler** 编译器 translates a high-level program into machine code **once, before it runs**.

- it reports **all** errors at compile time; once clean, it produces a stand-alone **executable** 可执行文件 that runs **without the compiler installed** and can be run many times.
- generally **faster at run time** (no translation while running), but tied to one CPU/OS — recompile for each platform.

## Interpreter

An **interpreter** 解释器 translates and runs a high-level program **one line at a time**, producing no executable.

- it reports an error **when it reaches that line**, then stops; you can fix it and continue — good for development.
- the **interpreter must be installed** to run the program; generally **slower** (each run re-translates), but easy to **port** across platforms.



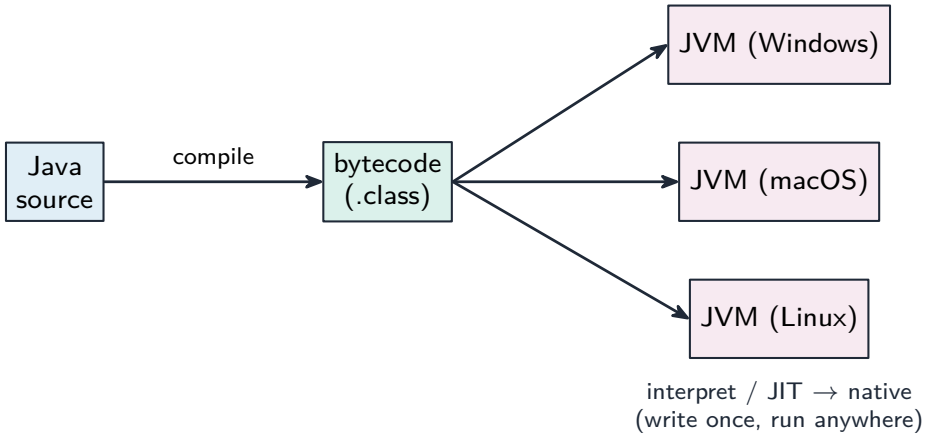

*A compiler translates once into a standalone program; an interpreter translates line by line, every run*

## Choosing between them

| Use a compiler when: | Use an interpreter when: |
|---|---|
| run-time speed matters | you want fast edit–run cycles |
| distributing to users without dev tools | writing cross-platform scripts |
| the program runs many times | the program is small or run once |
| | teaching beginners |

## Hybrid: Java

Java is compiled into **bytecode** 字节码 (a platform-independent intermediate form), which a **virtual machine** 虚拟机 (the JVM) then interprets — or uses **just-in-time compilation** 即时编译 to turn hot parts into native code. So errors are caught early, the bytecode runs anywhere with a JVM ("write once, run anywhere"), and long-running programs reach near-native speed. C# and Python use similar designs.

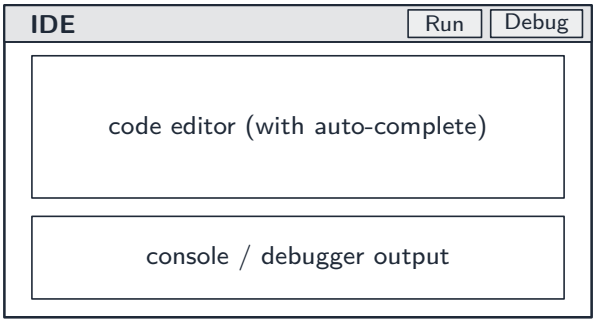



*Java compiles to portable bytecode that any JVM runs — write once, run anywhere*

**Worked example.** Java source is compiled to **bytecode**, which a JVM then interprets. Why use both, instead of compiling straight to machine code? A compiler produces machine code for **one** processor and operating system, so a program compiled on one machine will not run on another. Java's compiler instead targets a **virtual** machine, so the bytecode it produces is identical everywhere; each platform then supplies its own **JVM** to interpret that bytecode into its own native instructions. One compiled file therefore runs anywhere a JVM exists - "write once, run anywhere". The price is speed: interpreting bytecode is slower than running native code, which is why a real JVM also uses **JIT** compilation to turn frequently-run bytecode into native code while the program runs. Name **both** sides - the marks are for portability **bought at the cost of speed**.

# Integrated Development Environment (IDE)

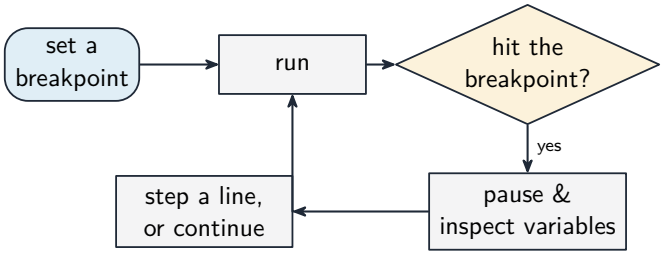
An **integrated development environment** 集成开发环境 (IDE) brings the tools to write, test and debug code into one application:




*An IDE bundles the editor, a Run button and a debugger*

- **source code editor** with **syntax highlighting** 语法高亮 (keywords, strings, comments in different colours), auto-indent and bracket matching.
- **auto-complete** 自动补全 — suggests names and shows function parameters as you type.
- **translator integration** — compile/run with one keystroke; errors shown inline.
- **debugger** 调试器 — set **breakpoints** 断点 to pause, **step through** line by line, and **inspect variables**.
- **version control** 版本控制 integration (git), **project management**, a help system, **refactoring** 重构 tools (safe renaming), and **unit test** 单元测试 integration.

An IDE speeds development by putting writing → running → debugging → fixing behind one interface. Common IDEs: Visual Studio, PyCharm, Eclipse, VS Code.




*A debugger: set a breakpoint, run, then pause to inspect variables and step through the code*

# Exam tips

- List the OS's jobs (memory, process, file, device and security management) — "manages resources" alone is too vague.
- Compare **compiler vs interpreter vs assembler**: what each translates and when errors are reported.
- Explain what an **IDE** provides (editor, debugger, error diagnostics, auto-complete).

# Security, privacy and data integrity

A-Level Computer Science

## Security, privacy and integrity — three different ideas

These sound alike but mean different things:

- **security** 安全 — protecting data from **unauthorised** 未授权 access, change or destruction.
- **privacy** 隐私 — an individual's right to **control who sees their personal data**, with consent and a clear purpose.
- **integrity** 完整性 — the data being **accurate and complete** — not corrupted or accidentally changed.

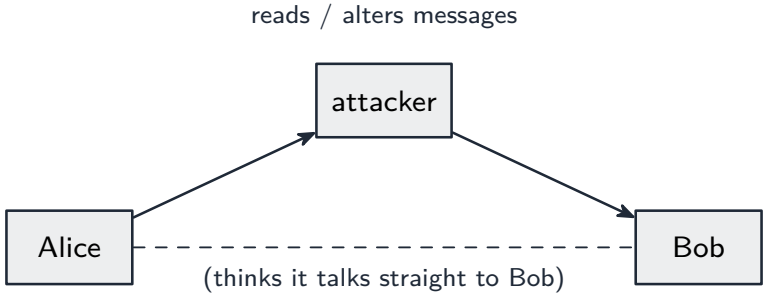
A file can be secure (only the right people can open it) but lack integrity (a typo corrupted it); or accurate but not private (anyone can read it). All three are needed.

## Why security matters

Two things to protect: the **data** itself (keep it confidential, intact and available) and the **computer system** (a compromised system can attack others, steal credentials, or be held to ransom).

## Threats from networks and the internet

Threats fall into three groups.




*A man-in-the-middle attacker sits between the two parties*

1. **Malware** 恶意软件 (malicious software) — harmful programs:

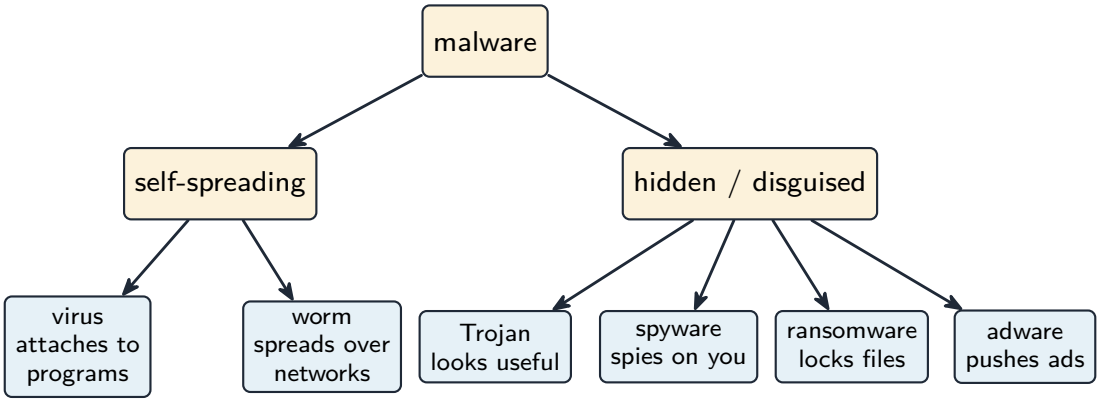
- **virus** 病毒 — self-copying code that attaches to other programs and spreads when they run.
- **worm** 蠕虫 — self-copying code that spreads over **networks** 网络 with no user action.
- **Trojan horse** 木马 — looks useful but hides malicious code.
- **spyware** 间谍软件 — secretly collects information (keystrokes, passwords).
- **ransomware** 勒索软件 — encrypts your files and demands payment.
- **adware** 广告软件 — pushes unwanted adverts.

2. **Tricking people** (social attacks):

- **phishing** 网络钓鱼 — fake emails/sites that trick users into giving credentials.
- **pharming** 域名欺骗 — redirects a user to a fake site even when they type the correct address.
- **social engineering** 社会工程 — tricking people into giving up information.

3. **Attacks on the network:**

- **hacking** 黑客入侵 by **hackers** 黑客 — unauthorised access, often via weak passwords or software flaws.
- **denial of service** 拒绝服务 (DoS/DDoS) — floods a server so real users cannot reach it.
- **eavesdropping** 窃听 — capturing data in transit (a risk on open Wi-Fi).
- **man-in-the-middle** 中间人攻击 — an attacker secretly relays or alters messages between two parties.




*Malware by behaviour: self-spreading (virus, worm) versus hidden/disguised (Trojan, spyware, ransomware, adware)*

# Security measures

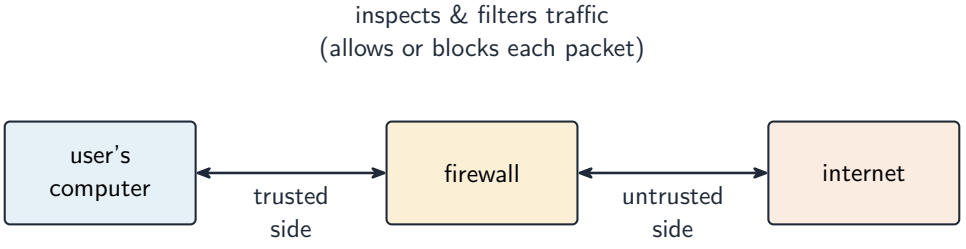
Measures protect both the **security of data** (against loss, theft or corruption) and the **security of the computer system** (its hardware, software and network).

## A standalone PC

- a strong password; **antivirus** kept up to date; prompt **software updates**; **backup** 备份 to separate media; full-disk **encryption** 加密; a locked screen.

## A networked PC

All the above, plus a **firewall** 防火墙, per-user **permissions** (admin rights only for admins), central management of **user accounts** 用户账户, and **audit logs** 审计日志 (who logged in, what they touched).




*A firewall sits between the user's computer and the internet*

## Across the internet

- **VPN** 虚拟专用网 — encrypts traffic between the user and the corporate gateway.
- **HTTPS / TLS** — encrypt web traffic.
- **digital signatures** 数字签名 — prove who sent a message and that it was not altered in transit.
- intrusion detection — watches traffic for known attack patterns.

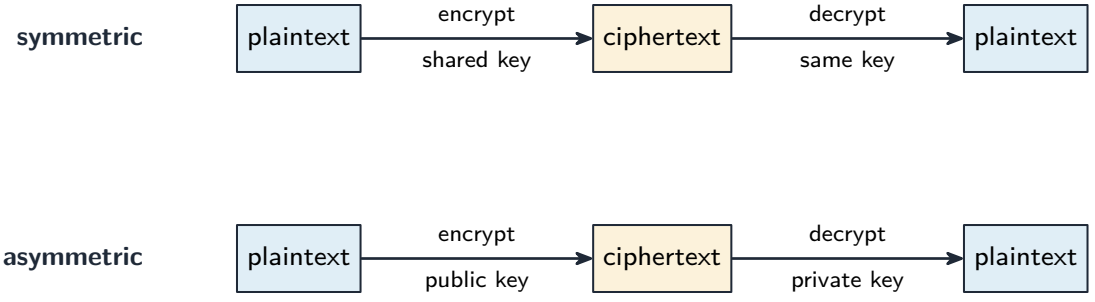
# Matching measures to threats

- **interception in transit** → **encrypt** the data (HTTPS, VPN). Intercepted ciphertext is useless without the key.
- **unauthorised access** → strong **authentication** 身份验证 (long passwords; **two-factor authentication** 双因素认证 with a phone code or key); user **authorisation** 授权; lock-out after failed logins.

- **malware** → **anti-virus software** and **anti-spyware** 反间谍软件 with real-time scanning; patching; avoid untrusted downloads.
- **phishing** → user training; email filtering; check the URL before entering credentials.
- **internal threats** → the **least-privilege** 最小权限 principle (give each user only what they need); auditing.
- **DDoS** → rate limiting and traffic filtering.

## Protecting the data itself

- **encryption** — turn **plaintext** 明文 into **ciphertext** 密文 with a key. **Symmetric encryption** 对称加密 (AES) uses one shared key; **asymmetric encryption** 非对称加密 (RSA) uses a **public key** 公钥 and a **private key** 私钥. Protects data at rest and in transit.
- **access control** 访问控制 — file permissions (read/write/execute) and **access rights** 访问权限, enforced by the OS.
- **authentication** — **authentication techniques** verify the user: something you know (password), have (token, phone), or are (**biometrics** 生物识别 — fingerprint, face, iris); strongest combined.
- **backups** — keep copies (some **off-site**) so loss or corruption is recoverable.
- **physical security** — locked server rooms, cable locks.




*Symmetric uses one shared key; asymmetric uses a public key to encrypt and a private key to decrypt*


*A security token shows a changing code for two-factor authentication ("something you have")*

*A fingerprint reader checks "something you are" — a feature of the person, not a password*

## Data integrity

Data has integrity when it is accurate and complete. Two techniques: **data validation** (catch bad data before storing) and **data verification** (confirm data was entered or transferred correctly).

## Validation — does the data make sense?

**Validation** 验证 checks data against sensible rules, automatically:

- **range check** — within limits (a month is 1–12).
- **limit check** — on the correct side of a single limit (e.g. age $\geq$ 18).
- **existence check** — the referenced item exists (e.g. a product code is in the table).
- **length check** — the right number of characters.
- **type / character check** — the right kind of data (a phone field allows only digits).
- **format check** — matches a pattern (an email must contain `@`).
- **presence check** — required fields are not empty.
- **check digit** 校验位 — an extra digit computed from the others (ISBN, card numbers) that spots transcription errors.
- **lookup check** and **consistency check** (e.g. delivery date $\geq$ order date).

Validation catches data that is wrongly **formatted**, but not data that is the right format yet factually wrong ("Bob" for "Bib").

## Verification — was the data entered or transferred correctly?

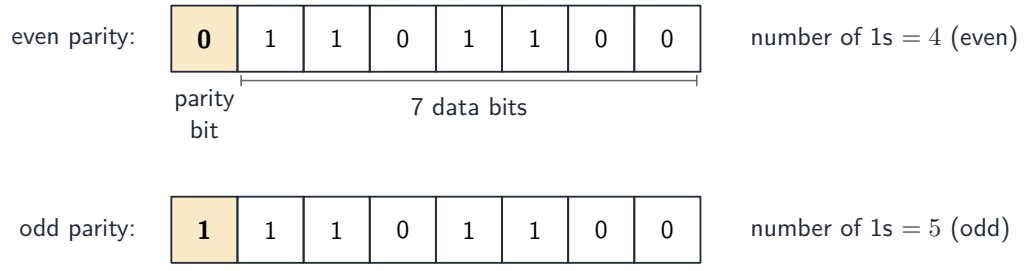
**Verification** 核对 checks the data was not changed in moving from one place to another.

During entry: **double entry** (type it twice and compare, as for a new password) or **visual** check.

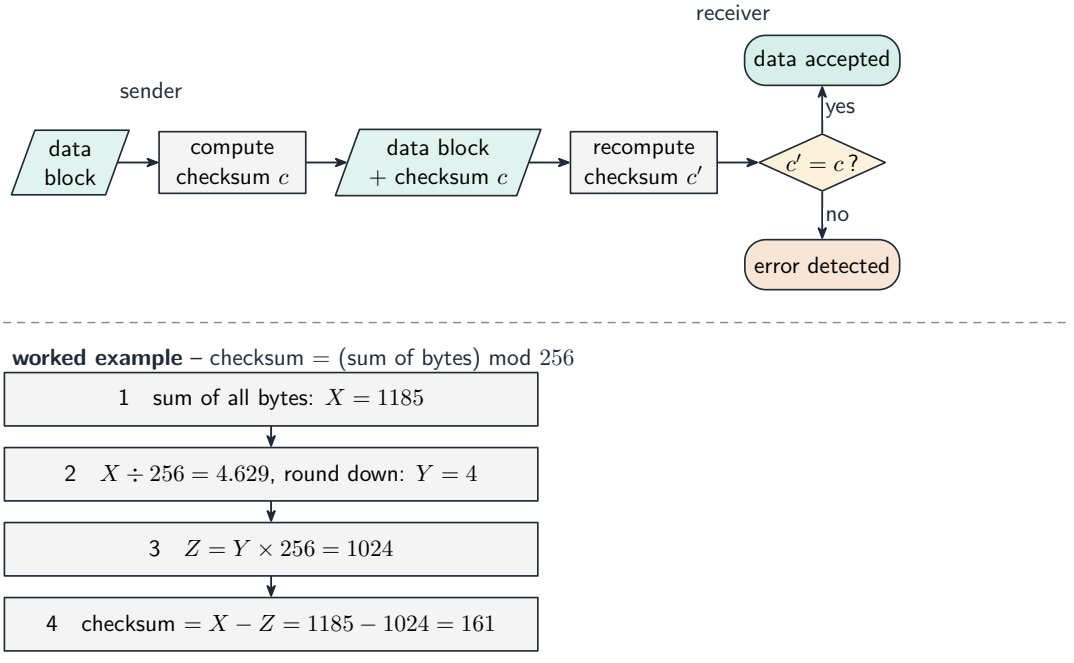
During transfer (bits can flip):

- **parity check** 奇偶校验 — an extra bit makes the number of 1s even (even parity) or odd. The receiver re-counts. Catches single-bit errors.
- **checksum** 校验和 — the sender sends a summary value of the data; the receiver recomputes it and compares.
- **cyclic redundancy check** 循环冗余校验 (CRC) — a stronger checksum using polynomial division, catching many more error types.

A **parity block check** 奇偶块校验 goes further and *locates* the error. Arrange the bytes in a grid: give each byte a **row** parity bit, then compute one extra **parity byte** whose bits are the **column** parity of the bytes above. A single flipped bit now fails **one row and one column** – their intersection pinpoints exactly which bit changed, so it can even be corrected.




*The parity bit is set to make the number of 1s even or odd*




**worked example** – checksum = (sum of bytes) mod 256

1 sum of all bytes: $X = 1185$

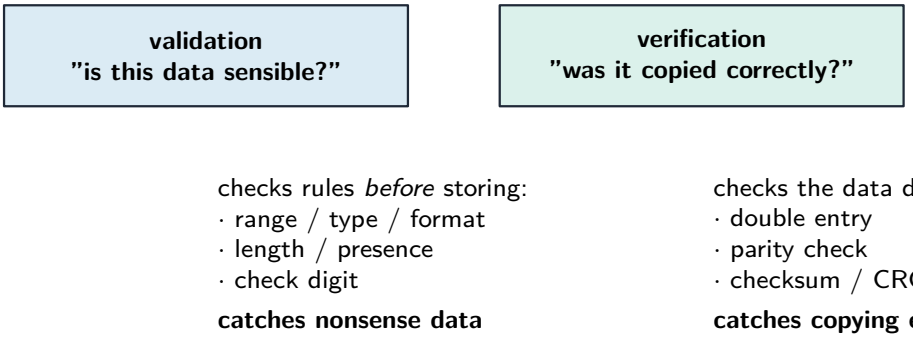
2 $X \div 256 = 4.629$, round down: $Y = 4$

3 $Z = Y \times 256 = 1024$

4 checksum $= X - Z = 1185 - 1024 = 161$

*Working out a checksum for a block of data*

Verification only proves what arrived matches what was sent — not that the data is correct, and not against deliberate tampering. **Validation** asks "is this sensible?"; **verification** asks "was this copied correctly?" — use both.




*Validation checks the data makes sense; verification checks it was copied without change*

**Worked example.** A user types their date of birth as `31/02/2009`, and types their email address twice. Which check catches which error, and what is the difference? **Validation** asks "is this data **sensible**?" - the computer tests it against a rule, and a format or range check rejects `31/02/2009` because February never has 31 days. **Verification** asks "was this data **entered correctly**?" - typing the email twice is **double entry**, and comparing the two copies catches a typing slip. The limit is what makes this a favourite question: validation can never tell you the data is **right**, only that it is **possible** - `01/02/2009` passes every validation rule even if the user was actually born on a different day. Say what each check **can** and **cannot** catch.

# Exam tips

- Keep the three ideas separate: **security** (keeping data safe), **privacy** (who may see it), **integrity** (keeping it correct).
- Match each **threat** (malware, hacking, phishing, interception) to a **measure** (firewall, encryption, authentication, access rights).
- Encryption protects **confidentiality, not integrity** — use a checksum, parity or check digit for integrity.
- Distinguish a **virus, worm and Trojan** and how each spreads.

# Ethics and Ownership

A-Level Computer Science

## Ethics for computing professionals

A **computing professional** is someone whose work — software, systems, networks, data — affects other people. Because the work is technical, others often cannot judge whether it was done well or honestly. So the profession follows shared **ethics** 伦理 (principles for good behaviour).

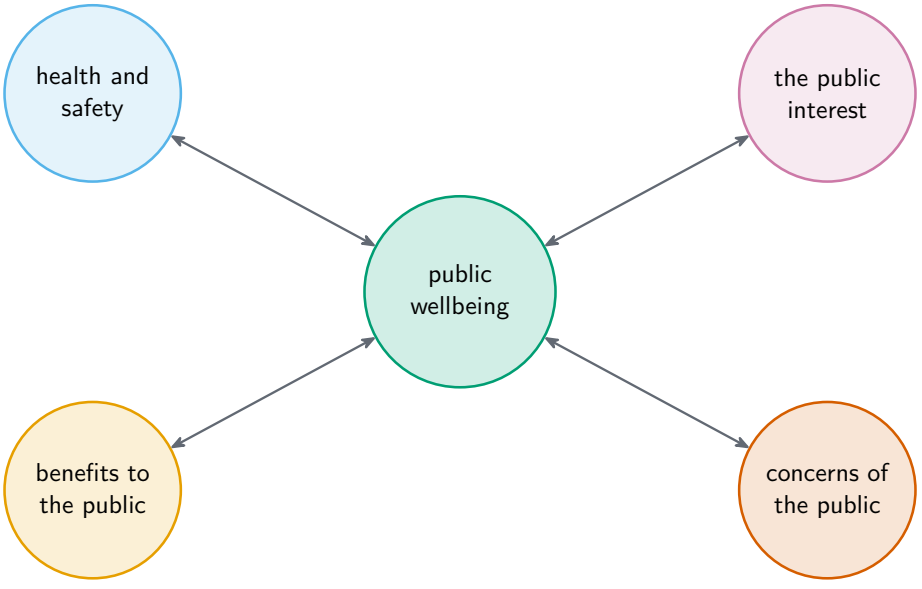
### Why ethics matters

- **trust** — users and employers trust professionals to act in their interest. Without that trust, software loses credibility.
- **impact** — software runs medical devices, banking, vehicles. Careless or dishonest work can hurt people.

Professional bodies (BCS, ACM, IEEE) publish codes of ethics for members.

*CCTV raises privacy concerns — one of the ethical issues a computing professional must weigh*

*Discarded electronics (e-waste) are a growing environmental cost of computing*




*Software development affects the public's wellbeing in several ways*

## Typical principles

- **public interest first** — protect the safety and welfare of those affected.
- **honesty and competence** — be honest about your skills; don't claim expertise you lack.
- **confidentiality** 保密性 — protect clients' and employers' private information.

- **avoid conflicts of interest** 利益冲突 — don't take work where your interest clashes with the client's.
- keep your skills current; respect **intellectual property** 知识产权 and **privacy** 隐私; treat colleagues fairly.

### Acting ethically vs unethically

Acting **ethically** protects users, strengthens reputation, reduces legal risk, and builds trust. Acting **unethically** (skipping testing, hiding bugs, misusing data) can harm real users, lead to dismissal or legal action, damage reputation, and erode trust in technology generally.

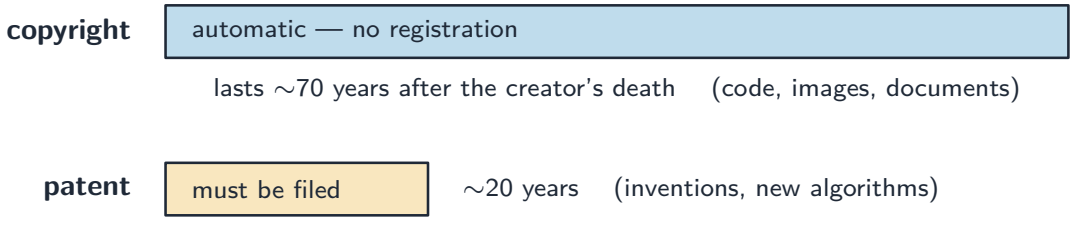
When you face a borderline decision: identify whose interests are affected, check the code of ethics and the law, weigh the consequences, ask a trusted senior, and choose the option that protects users above short-term convenience.

**Worked example.** Your team's new AI hiring tool sorts CVs ten times faster, but you notice it rejects more older applicants. Shipping it pleases your manager, but it treats one group unfairly. The ethical choice is to **hold it back until the bias is fixed** — public interest and fairness come before short-term convenience.

## Copyright

**Copyright** 版权 is the legal right of the creator of an original work to control how it is **copied, distributed, modified and performed**. It applies automatically (no registration) to source code, software, documents, images, audio and video.

Without copyright, anyone could copy software freely, the developer would not be paid, and plagiarism would be legal. With copyright, developers can earn from their work (encouraging more software), users know who made it, and re-use happens on the developer's terms through licensing. Copyright lasts a long time (often 70 years after the creator's death). General ideas and algorithms are not covered by copyright but may be covered by a **patent** 专利.




*Copyright is automatic and long-lasting; a patent must be filed and lasts about 20 years*

# Software licences

A **software licence** 软件许可证 is a contract granting permission to use software on the owner's terms; choosing and applying one is called **software licencing**.

## Commercial (proprietary)

- **commercial software** is sold: you **buy a licence**; the software is used only within its terms.
- the source code is not given (a **proprietary** 专有 product); you cannot modify or redistribute it.
- examples: Microsoft Office, Adobe Photoshop, most games.

Used when the developer wants **revenue** per user and to **keep control** of the code.

## Open-source

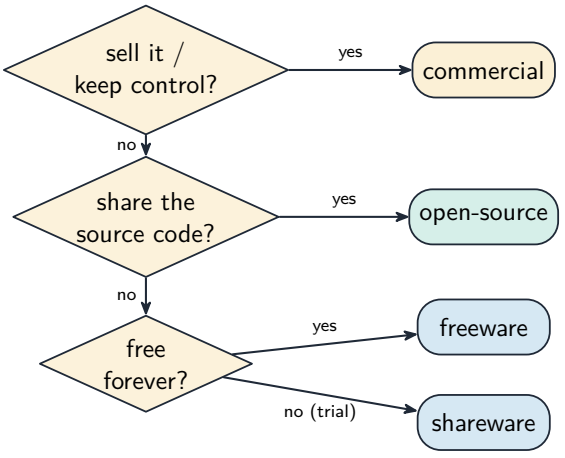
- the source code is **public**; users can read, modify and redistribute it (**open-source** 开源).
- **permissive** licences (MIT, BSD) allow almost any use; **copyleft** 著佐权 licences (GPL) require that modified versions are released under the same licence ("share-alike").
- the **Free Software Foundation** (FSF) and the **Open Source Initiative** (OSI) promote and approve open-source licences.
- examples: Linux, Python, Apache.

Used when the developer wants the software **widely used and improved by the community**.

## Freeware and shareware

- **freeware** 免费软件 — free of charge, no source code, may be redistributed but not modified (Acrobat Reader, WhatsApp).
- **shareware** 共享软件 — free for a trial period, then you pay to keep using it; no source code.

| Type | Cost | Source | Redistribute | Modify |
|---|---|---|---|---|
| Commercial | Paid | No | No | No |
| Open-source | Free | Yes | Yes | Often, with conditions |
| Freeware | Free | No | Yes | No |
| Shareware | Free trial, then paid | No | Sometimes | No |




*Choosing a licence from the developer's goal*

To justify a licence choice, link it to the developer's goal (revenue, reach, community), the user's needs (cost, customising), and the use case.

# Artificial Intelligence (AI)

**Artificial intelligence** 人工智能 builds systems that do tasks once thought to need **human intelligence** — recognising speech and images, translating, playing games, driving.

Most modern AI uses **machine learning** 机器学习 — algorithms that improve at a task by learning patterns from large amounts of data, instead of being programmed step by step. **Deep learning** 深度学习, using **neural networks** 神经网络 with many layers, is the leading approach today.

## Everyday examples

AI tasks split into two kinds — **understanding input**, and **producing output or decisions**.

Understanding input:

- **speech recognition** 语音识别 — spoken words to text (voice assistants).
- **image recognition** 图像识别 — finding objects, faces or text in images.

Producing output or decisions:

- **machine translation** 机器翻译 — automatic translation between languages.
- **recommendation systems** 推荐系统 — suggesting products, videos or music.
- **autonomous vehicles** 自动驾驶汽车 and robots.

A common exam scenario: a program reads a label with a camera, translates it, and reads it aloud — using **optical character recognition** 光学字符识别 to find the words, machine translation to convert them, and **text-to-speech** 文本转语音 for the audio.

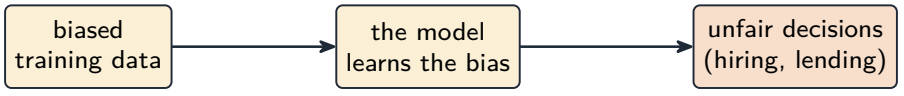

*A common scenario: OCR → machine translation → text-to-speech reads a foreign label aloud*

## Benefits

- **accessibility** — speech/image AI helps users with impairments; translation helps non-native speakers.
- **productivity** — automating repetitive tasks frees people for creative work.
- **decision support** — AI spots patterns in huge datasets (medical diagnosis, fraud detection).
- **always available**, and **personalised** to each user.

## Concerns

- **bias** 偏见 — unfair patterns in the training data become unfair AI decisions (hiring, lending).
- **job displacement** — AI may replace some roles.
- **privacy** — training often uses large amounts of personal data.
- **transparency** — large models are "black boxes", hard to explain.
- **accountability** — when AI is wrong, who is responsible: developer, user, or operator?
- **misuse** — deepfakes, misinformation, surveillance.




*How bias gets into AI: biased data → a biased model → unfair decisions*

Professionals must **understand the limits** of the AI they build, **inform users**, and **reduce harm**.

# Exam tips

- Answer ethics questions against a **professional code of conduct** (public interest, competence, honesty), not personal opinion.
- Distinguish **copyright** (protects the expression) from a **patent** (protects an invention).
- Compare software licences: proprietary, open-source, freeware, shareware and FOSS.

# Databases

A-Level Computer Science

## File-based storage and its limits

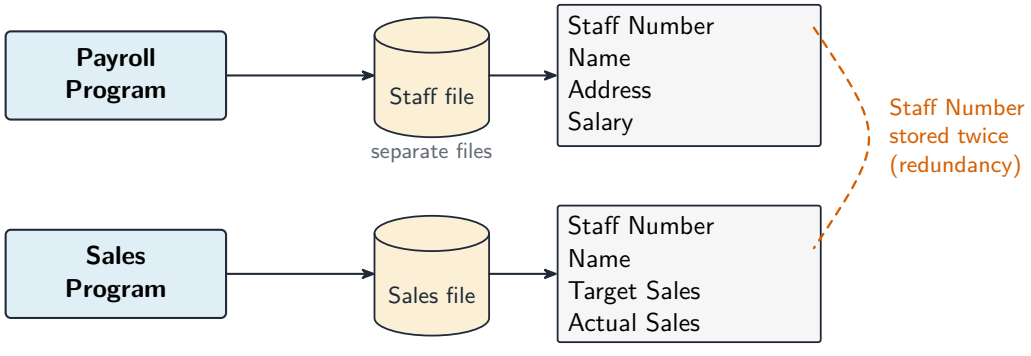
Before databases, programs stored data in **flat files** 平面文件 — usually one file per program. This is fine for small data but breaks down at scale.


*File-based storage keeps data in separate files, like papers in a filing cabinet — hard to search and easy to duplicate*

## Limitations

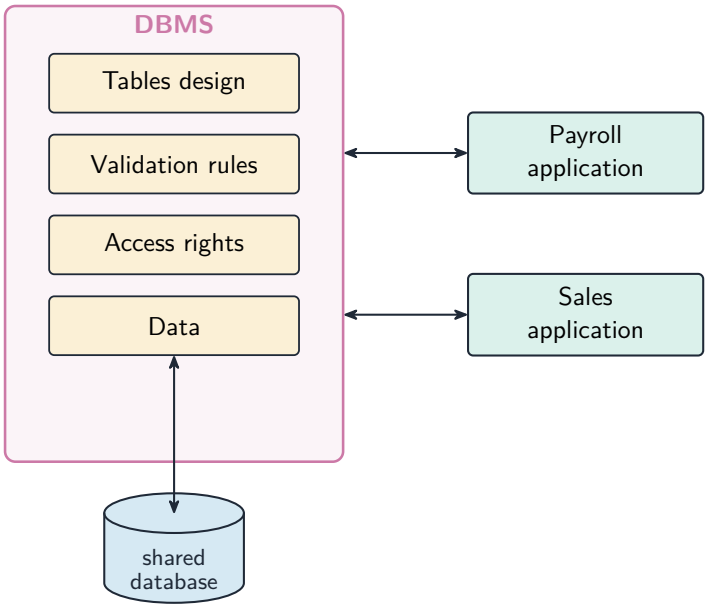
- **data redundancy** 数据冗余 — the same data (a customer's address) is held in several files.
- **data inconsistency** 数据不一致 — redundant copies updated separately get out of sync.
- **data dependence** — programs are tied to the file format; change the format and every program must be rewritten.
- hard to enforce **integrity** 完整性, hard to share safely, weak querying, and weak per-field security.

*The files themselves are stored on devices such as hard disk drives*




*The file-based approach: each program keeps its own files*

A **relational database** 关系数据库 fixes these by storing data in **tables** managed by one piece of software (the DBMS) that all programs use.



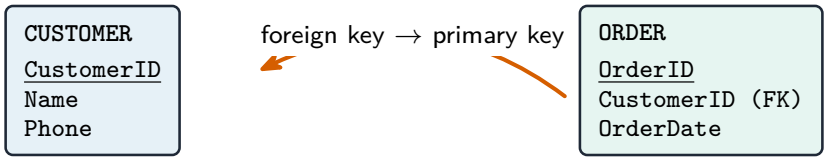

*The database approach: one DBMS serves all the programs*

# Relational model — terms

- **table** 表 (relation) — a grid of rows and columns; one table per type of **entity** 实体 (e.g. `CUSTOMER`).
- **record** 记录 (row, also called a **tuple** 元组) — one row; one instance of the entity.
- **field** 字段 (column, also called an **attribute** 属性) — one column; one piece of information about each record.
- **primary key** 主键 — a field (or fields) that **uniquely identifies** each record; never null or duplicated.
- **foreign key** 外键 — a field whose value matches the primary key of another table, linking the two.
- **composite key** 复合键 — a primary key made of two or more fields together.
- **candidate key** 候选键 — any field(s) that could be the primary key.
- **secondary key** 次键 — a non-primary field that is indexed for fast searching.
- **indexing** 索引 — building an index on a field so look-ups and joins run faster.
- **referential integrity** 参照完整性 — every foreign-key value must match an existing primary key (no orphan records).

A table is written in shorthand with the primary key underlined and foreign keys noted:

```
CUSTOMER(CustomerID, Name, Phone)
ORDER(OrderID, CustomerID, OrderDate)   -- CustomerID is FK → CUSTOMER
```

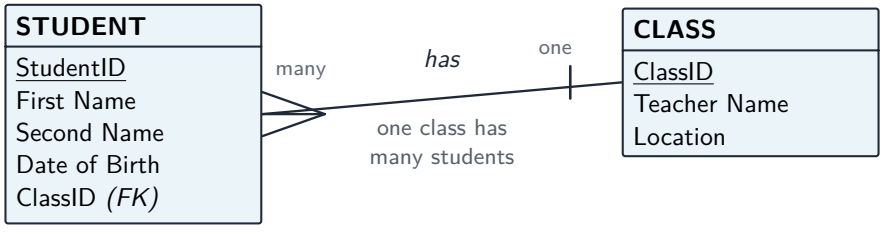



*A foreign key links two tables: ORDER.CustomerID matches the primary key CUSTOMER.CustomerID*

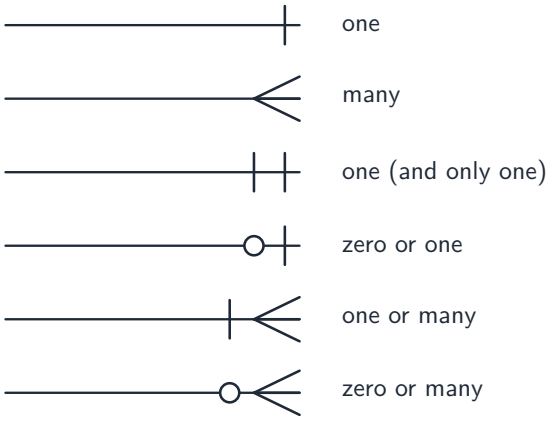
# Entity-relationship (E-R) diagrams

An **entity-relationship diagram** 实体关系图 shows the structure: each entity is a rectangle, each relationship a line, with the **cardinality** 基数 marked at each end:

- **one-to-one** (1:1).
- **one-to-many** 一对多 (1:M) — each Customer has many Orders; each Order has one Customer.
- **many-to-many** (M:N) — Students take many Courses, and Courses have many Students.




*An E-R diagram: one class has many students*




*Crow's-foot symbols for the cardinality of a relationship*

A many-to-many relationship cannot be stored directly. Break it into two one-to-many relationships through a **link table** 连接表 holding the two foreign keys:

```
ENROLMENT(StudentID, CourseID, EnrolmentDate)
```


a link table turns many-to-many into two one-to-many relationships

*A link table resolves a many-to-many relationship into two one-to-many relationships*

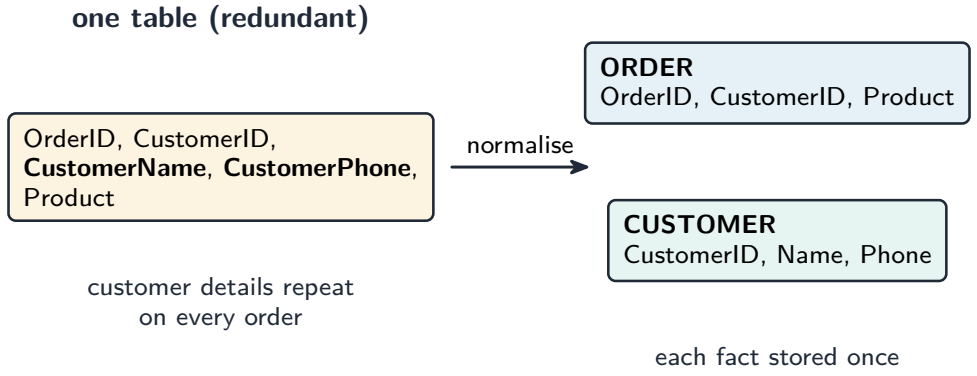
# Normalisation

**Normalisation** 规范化 organises tables to **cut redundancy and inconsistency**, going through **normal forms** 范式 in order.

- **First normal form (1NF)** — every field holds a single (**atomic** 原子) value, with no repeating groups, and a primary key.
- **Second normal form (2NF)** — in 1NF, and every non-key field depends on the **whole** primary key (only matters for a composite key).
- **Third normal form (3NF)** — in 2NF, and every non-key field depends **only** on the primary key, not on another non-key field (no **transitive dependency** 传递依赖).

A 3NF design stores each fact once, so insert/update/delete anomalies disappear. The trade-off is more tables and more joins. Aim for 3NF.

To produce a 3NF design: find the entities and their attributes; choose a primary key for each; split repeating/non-atomic fields (1NF); split fields depending on part of a composite key (2NF); split fields depending transitively on the key (3NF); add foreign keys for the relationships.




*Normalisation removes redundancy by splitting repeated data into its own table*

**Worked example.** The table `ORDER(OrderID, CustomerID, CustomerName, ProductID, Quantity)` has the composite primary key `(OrderID, ProductID)`. Normalise it to 3NF. Test each non-key field against the key. `Quantity` depends on **both** `OrderID` and `ProductID`, which is fine. But `CustomerID` depends on **`OrderID` alone** - only *part* of the composite key. That is a **partial dependency**, so the table is not in **2NF**. Split it into `ORDER_LINE(OrderID, ProductID, Quantity)` and `ORDER(OrderID, CustomerID, CustomerName)`. Now test 3NF: in that new `ORDER` table, `CustomerName` depends on `CustomerID`, which is **not** the key - a **transitive dependency**. Split again: `ORDER(OrderID, CustomerID)` and `CUSTOMER(CustomerID, CustomerName)`. Name the **dependency** that breaks each form (partial breaks 2NF, transitive breaks 3NF); "it has repeated data" describes the symptom and earns nothing.

# Database Management System (DBMS)

A **DBMS** 数据库管理系统 manages the database centrally. Features that fix the file-based limits:

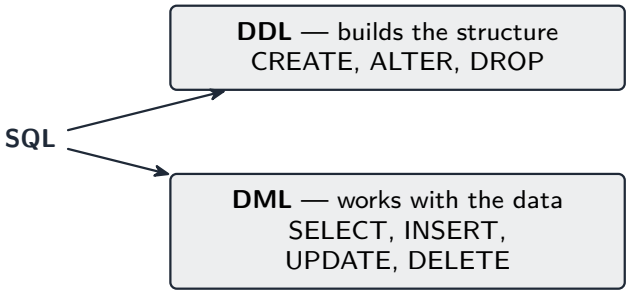
- **data dictionary** 数据字典 — a description of every table, field, type and key; programs query it instead of hard-coding the structure.
- redundancy/consistency control — each fact stored once.

- **concurrent access** 并发访问 control — locks and transactions let many users work at once.
- **backup** 备份 and recovery; security and per-user permissions.
- **integrity rules** — keys, unique and range constraints, enforced centrally.
- **transactions** 事务 — a group of operations that all succeed or all fail.
- **views** 视图 — virtual tables that show each user "their" slice of the data.
- **data management** 数据管理 and **data modelling** 数据建模 — control how data is stored and define its structure as a **logical schema** 逻辑模式 (the logical design, independent of physical storage).
- **data integrity** 数据完整性 and **data security** 数据安全 — enforce correctness and control access centrally.
- a **query processor** 查询处理器 runs queries; a **developer interface** 开发者接口 gives tools and APIs for building applications.

Its tools include a data-dictionary editor, a query builder, a forms builder, a report generator, user management, and an SQL editor.

# DDL and DML

**SQL** 结构化查询语言 (Structured Query Language) has two halves:




*DDL builds the database structure; DML works with the data*

- **Data Definition Language** 数据定义语言 (DDL) — creates or changes the **structure** (tables, keys, constraints).
- **Data Manipulation Language** 数据操纵语言 (DML) — works with the **data** (insert, update, delete, **query** 查询).

## DDL basics

```
CREATE TABLE CUSTOMER (
  CustomerID INTEGER PRIMARY KEY,
  Name VARCHAR(50) NOT NULL,
  Phone VARCHAR(20)
);
```

Add a foreign key:

```
CREATE TABLE ORDER (
  OrderID INTEGER PRIMARY KEY,
  CustomerID INTEGER,
  OrderDate DATE,
  FOREIGN KEY (CustomerID) REFERENCES CUSTOMER(CustomerID)
);
```

Modify and drop:

```
ALTER TABLE CUSTOMER ADD Email VARCHAR(100);
DROP TABLE CUSTOMER;
```

Common types: `INTEGER`, `REAL`, `VARCHAR(n)`, `CHAR(n)` (also `CHARACTER(n)`), `DATE`, `TIME`, `BOOLEAN`, `DECIMAL(p, s)`.

## DML basics

Query with `SELECT`:


*A SELECT query returns only the rows that match its condition*

```
SELECT Name, Phone
FROM CUSTOMER
WHERE City = 'London'
ORDER BY Name ASC;
```

`SELECT` lists fields, `FROM` names the table, `WHERE` filters rows, `ORDER BY` sorts.

A **join** 连接 combines two tables using a foreign-key relationship:

```
SELECT C.Name, O.OrderDate
FROM CUSTOMER C INNER JOIN ORDER O
  ON C.CustomerID = O.CustomerID
WHERE O.OrderDate >= '2024-01-01';
```

**Aggregate functions** 聚合函数 (`COUNT`, `SUM`, `AVG`, `MIN`, `MAX`) are often used with `GROUP BY`:

```
SELECT CustomerID, COUNT(*) AS NumOrders
FROM ORDER
GROUP BY CustomerID;
```

Insert, update, delete:

```
INSERT INTO CUSTOMER (CustomerID, Name, Phone)
VALUES (101, 'Ada Lovelace', '020-1234-5678');

UPDATE CUSTOMER SET Phone = '020-9999-0000' WHERE CustomerID = 101;

DELETE FROM CUSTOMER WHERE CustomerID = 101;
```

Always put a `WHERE` clause on `UPDATE` and `DELETE`, or the change hits **every** row.

### Tips for exam SQL

- use the exact table and field names from the question.
- quote strings with single quotes (`'Smith'`); don't quote numbers.
- comparisons: `=`, `<`, `>`, `<=`, `>=`, `<>`.
- `LIKE 'A%'` matches anything starting with A (`%` = any string, `_` = one character); `IN (1,2,3)`; `BETWEEN 10 AND 20`.
- combine conditions with `AND` / `OR` / `NOT`, and end each statement with a semicolon.

## Exam tips

- Define the terms exactly: entity, attribute, **primary key, foreign key**, and the relationship types (1:1, 1:many, many:many).
- Give a reason at each normal form: **1NF** (no repeating groups), **2NF** (no partial dependency), **3NF** (no non-key dependency).
- Explain what a **DBMS** provides (data independence, security, integrity, concurrent access).
- Distinguish **DDL** (define the structure) from **DML** (query and change the data).

# Algorithm Design and Problem-solving

A-Level Computer Science

## Computational thinking

**Computational thinking** 计算思维 is the set of mental tools for **analysing a problem and designing a solution a computer can run**. Two key ones are abstraction and decomposition.


*Computational thinking breaks a big problem into smaller, easier parts — like solving a jigsaw*

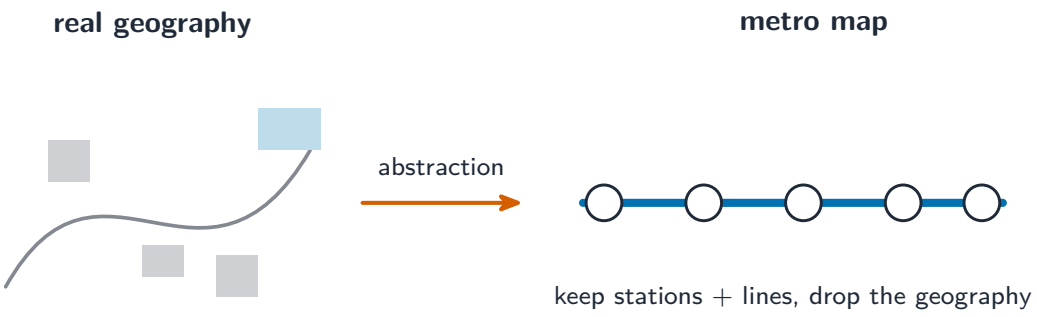
### Abstraction

**Abstraction** 抽象 means **keeping the essential features** of a problem and **ignoring the irrelevant detail**, giving a simpler model.

Examples:

- a **train-network map** keeps the stations and lines but drops the geography.
- a **class** in object-oriented programming keeps only the attributes and methods the system needs.
- a **function** hides a piece of work behind a name.

A full model of any real problem would be too big to reason about, so abstraction is essential.



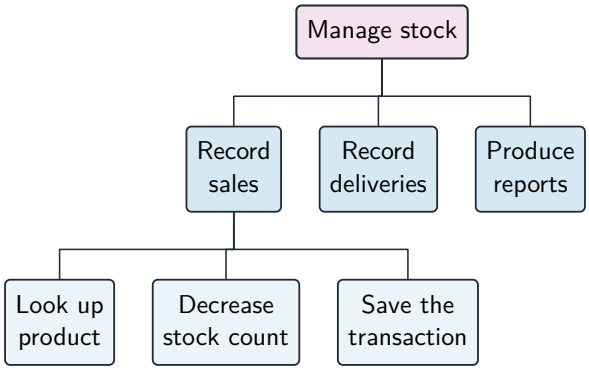

*Abstraction keeps the essentials (stations and lines) and drops irrelevant detail (the geography)*

## Decomposition

**Decomposition** 分解 means **breaking a large problem into smaller sub-problems**, each easier to solve and tackled one at a time.

1. find the main parts of the task.
2. break each into smaller sub-tasks.
3. continue until each is small enough to design directly.
4. solve the small tasks and combine them.

For stock control: "manage stock" → "record sales", "record deliveries", "produce reports" → ("record sales") "look up product", "decrease stock count", "save the transaction". Decomposition makes big problems manageable, lets a team divide the work, and gives modular code — each module becomes a **procedure** 过程 or function.




*Decomposing a program into modules and sub-modules*

# Algorithms

An **algorithm** 算法 is a **solution expressed as a sequence of defined steps**. Each step is **unambiguous** 无歧义 (one meaning), **deterministic** 确定性 (same input → same output), **finite** (the steps end), and **effective** (each can be done). An algorithm says *what to do*, independent of the programming language used to implement it.

# Identifier table

When you start an algorithm, list every piece of data in an **identifier table** 标识符表 — its **variable** 变量 name, **data type** 数据类型, and description:

| Variable name | Data type | Description |
|---|---|---|
| `Category` | `STRING` | the product category |
| `SaleDate` | `DATE` | when the item was sold |
| `ItemCost` | `REAL` | cost of the item |
| `InStock` | `BOOLEAN` | `TRUE` if in stock |

Use **descriptive** names (`ItemCost`, not `x`); common types are `INTEGER`, `REAL`, `STRING`, `CHAR`, `BOOLEAN`, `DATE`, plus arrays. The table forces you to name every piece of data before writing code.

**list every piece of data first** - name, type, description

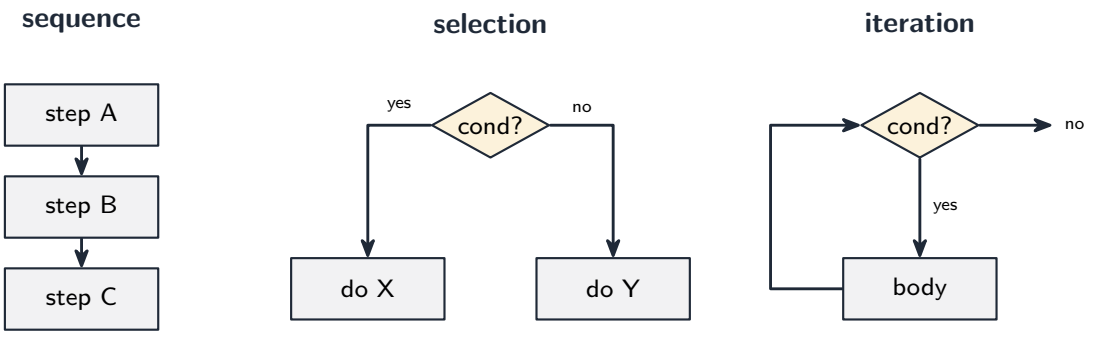
| name | data type | description |
|---|---|---|
| `ItemCost` | `REAL` | cost of the item |
| `Category` | `STRING` | the product category |
| `SaleDate` | `DATE` | when the item was sold |
| `InStock` | `BOOLEAN` | TRUE if in stock |

use **descriptive** names (`ItemCost`, not `x`) - the table forces you to plan the data

*An identifier table names every piece of data before you write code*

# Pseudocode — the three basic constructs

**Pseudocode** 伪代码 is a structured, language-neutral way to describe algorithms.




*The three building blocks of any algorithm: sequence, selection and iteration*

## 1. Sequence

Steps run one after another (**sequence** 顺序):

```
INPUT Name
INPUT Age
OUTPUT "Hello", Name
```

## 2. Selection

A choice of which steps run, based on a condition (**selection** 选择):

```
IF Age >= 18 THEN
    OUTPUT "Adult"
ELSE
    OUTPUT "Minor"
ENDIF
```

For more options, use `CASE OF ... ENDCASE`.

## 3. Iteration

Repeating a block (**iteration** 迭代, a **loop** 循环):

```
FOR i ← 1 TO 10
    OUTPUT i
```

```
NEXT i
```

A **WHILE** loop tests the condition **before** each pass (may run zero times); a **REPEAT...UNTIL** loop tests **after** each pass (always runs at least once).

### Common operations

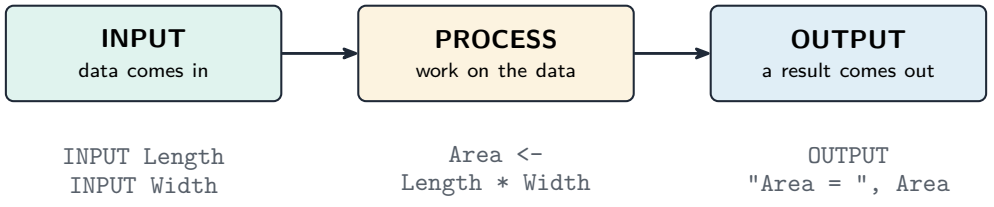
- **assignment** 赋值: `x ← 5` (an arrow; `=` is for comparison).
- input/output: `INPUT variable`, `OUTPUT expression`.
- comparisons `=`, `<>`, `<`, `>`, `<=`, `>=`; logic `AND`, `OR`, `NOT`.
- arithmetic `+ - * /`, plus `DIV` (integer division) and `MOD` (remainder).
- strings: `LENGTH`, `LEFT`, `RIGHT`, `MID`, and `&` for **concatenation** 拼接 (joining).

### Input → Process → Output

Every program follows this shape:

```
INPUT Length
INPUT Width
Area ← Length * Width
OUTPUT "Area = ", Area
```

Listing the inputs and outputs first makes the algorithm cleaner.




*Every program follows the Input, Process, Output shape*

## Three notations

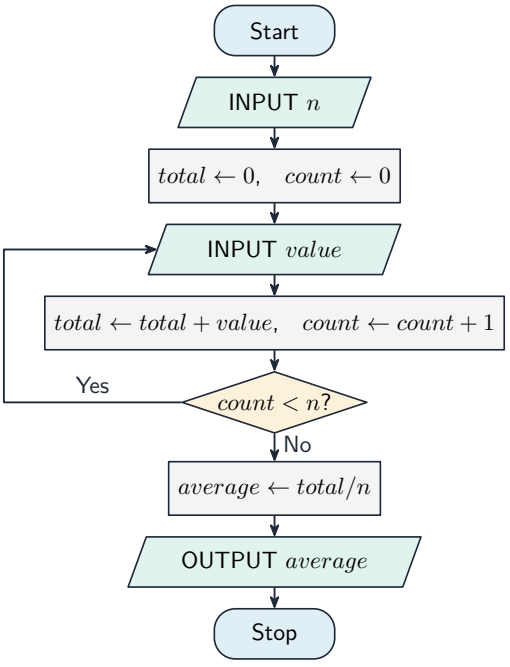
The same algorithm can be written three ways.

- **structured English** 结构化英语 — natural language with indentation and fixed keywords; good for a high-level description.
- **flowchart** 流程图 — a diagram with standard shapes:

| Shape | Meaning |
|---|---|
| Rounded rectangle | Start / Stop |
| Parallelogram | Input / Output |
| Rectangle | Process |
| Diamond | Decision |
| Arrow | Flow of control |

- **pseudocode** — the keyword notation above; closest to code.

You should be able to convert between any pair: each `IF` is a decision diamond, each loop is a back-arrow, and a sequence is stacked rectangles.




*A flowchart for averaging a list of numbers, using the standard shapes*

# Stepwise refinement

**Stepwise refinement** 逐步求精 starts with a high-level outline and **expands each step** until it is small enough to code. For an average of $n$ numbers:

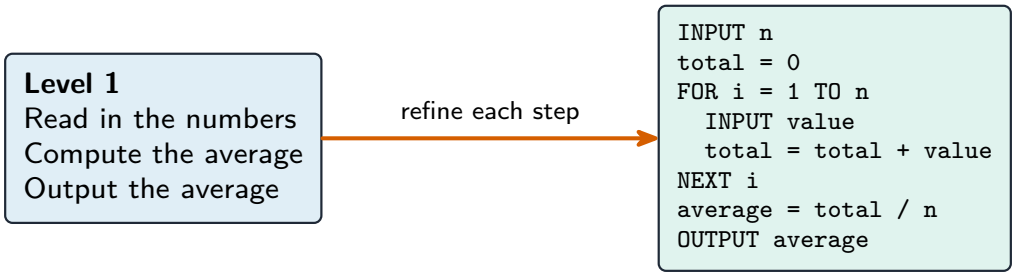
**Level 1**:

```
Read in the numbers
Compute the average
Output the average
```

**Level 2**:

```
INPUT n
total ← 0
FOR i ← 1 TO n
    INPUT value
    total ← total + value
NEXT i
average ← total / n
OUTPUT average
```

Each refinement keeps the previous structure and adds detail.




*Stepwise refinement: expand each high-level step into detailed pseudocode*

# Logic statements

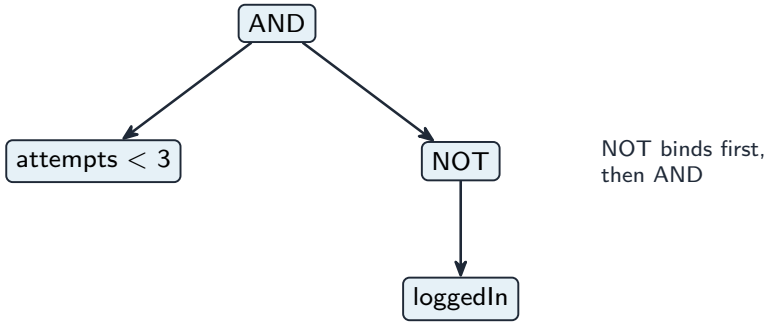
A **logic statement** 逻辑语句 is a **Boolean** 布尔 condition that controls branching, built from comparisons (`x > 10`), connectives (`AND`, `OR`, `NOT`) and brackets. Use it as the condition of `IF`, `WHILE` or `REPEAT...UNTIL`:

```
WHILE attempts < 3 AND NOT loggedIn DO
    INPUT password
    IF password = correctPassword THEN
        loggedIn ← TRUE
    ELSE
        attempts ← attempts + 1
    ENDIF
ENDWHILE
```

**Precedence** 优先级 (highest to lowest): `NOT`, then `AND`, then `OR`. Use brackets when unsure. Common mistakes:

- `a = 1 OR 2` is wrong — write `a = 1 OR a = 2`.
- `NOT a > 5` means `NOT (a > 5)`, i.e. `a <= 5`.

- `NOT (A AND B)` is the same as `(NOT A) OR (NOT B)` (**De Morgan's law** 德摩根定律) — handy for simplifying conditions.




*Precedence: NOT binds to loggedIn first, then AND combines the two sides*

**Worked example.** Write an identifier table and pseudocode to read 10 numbers and output the largest. The **identifier table** names each variable with its data type and purpose: `Count : INTEGER` (loop counter), `Num : REAL` (the number just read), `Max : REAL` (largest so far).

```
Max ← -999999
FOR Count ← 1 TO 10
    INPUT Num
    IF Num > Max THEN
        Max ← Num
    ENDIF
NEXT Count
OUTPUT Max
```

The design decision carrying the marks is **initialising `Max`**: it must start **lower than any possible input** - or, safer still, be set to the **first** number read. Initialise it to `0` and the algorithm wrongly returns 0 for a list of negative numbers, a bug your trace only exposes if the test data include a negative.

# Exam tips

- Define an **algorithm** as an unambiguous, finite, deterministic sequence of steps, independent of language.
- Use the three constructs correctly — **sequence, selection, iteration** — and keep an identifier table with data types.
- Break a problem down by **decomposition and abstraction**, then stepwise refinement.

- Write pseudocode that would actually run: declare variables and follow the exam's pseudocode style.

# Data Types and Structures

A-Level Computer Science

## Choosing data types

Every variable needs a **data type** 数据类型 — the kind of value it holds and the operations allowed:

- `INTEGER` — a whole number (`42`, `-7`). For counts, indexes, IDs.
- `REAL` — a number with a fractional part (`3.14`). For money, measurements.
- `STRING` — characters in quotes (`"Hello"`). For text.
- `CHAR` — a single character (`'A'`).
- `BOOLEAN` — `TRUE` or `FALSE`. For flags.
- `DATE` — a calendar date.

Pick the smallest precise type that fits: `INTEGER` for whole counts, `BOOLEAN` for flags (not the strings `"yes"`/`"no"`).

## Records

A **record** 记录 (a **record structure** 记录结构) holds **several fields of different types** under one name — useful when several values describe one thing.

```
TYPE TStockItem
    DECLARE ItemID : INTEGER
    DECLARE Category : STRING
    DECLARE ItemCost : REAL
    DECLARE InStock : BOOLEAN
ENDTYPE
```

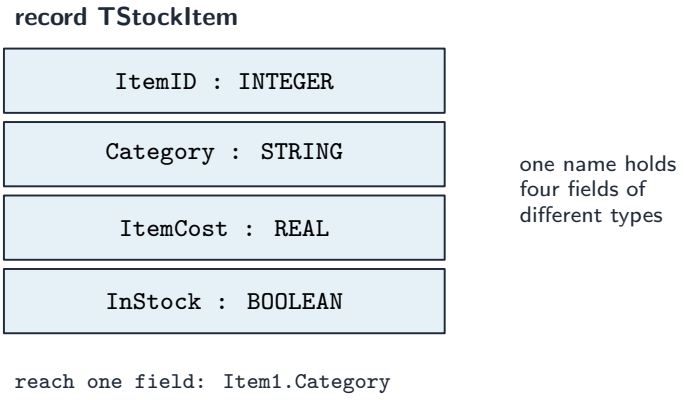
This defines the **type** `TStockItem`; declare variables of it:

```
DECLARE Item1 : TStockItem
DECLARE Items : ARRAY[1:100] OF TStockItem
```

Use dot notation to reach each **field** 字段:

```
Item1.Category ← "Fruit"
OUTPUT Item1.Category, " costs ", Item1.ItemCost
```

Use a record when values **always belong together** (a customer, a stock item); use separate variables for unrelated values.




*A record holds several fields of different types under one name*

# Arrays

An **array** 数组 is an ordered collection of items of the **same type**, under one name, reached by an **index** 索引.

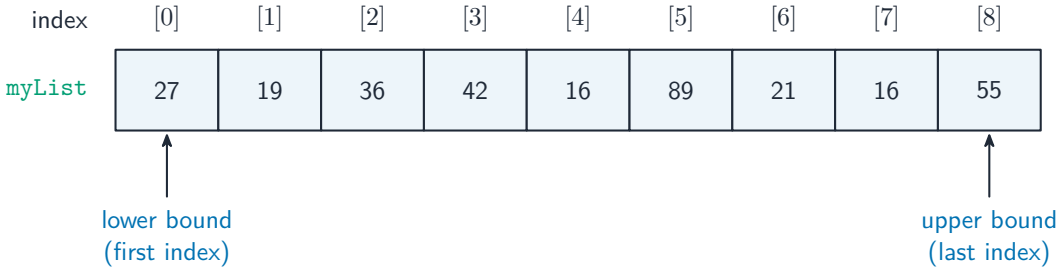
- **element** 元素 — one item in the array.
- **bounds** 边界 — the lowest and highest valid indices.
- **dimension** 维度 — 1-D (a list), 2-D (a table), etc.

## 1-D arrays

```
DECLARE Names : ARRAY[1:5] OF STRING
Names[3] ← "Cara"
OUTPUT Names[3]
```

Process every element with a `FOR` loop:

```
FOR i ← 1 TO 5
    OUTPUT Names[i]
NEXT i
```

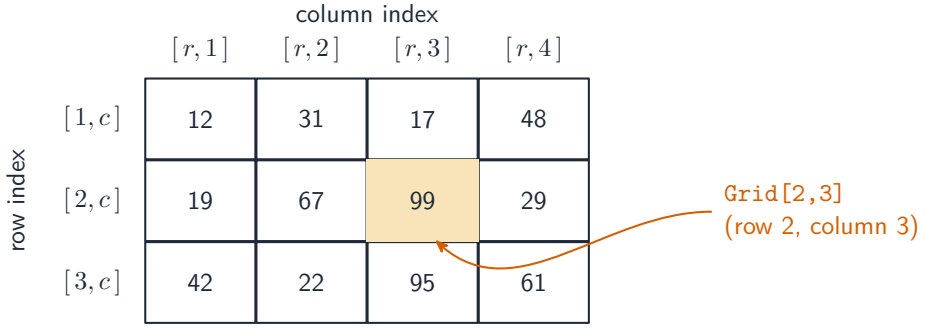



*A 1-D array (a list) with indices and bounds*

## 2-D arrays (2D array)

```
DECLARE Grid : ARRAY[1:3, 1:4] OF INTEGER
Grid[2, 3] ← 99
```

The first index is the row, the second the column. Use nested loops to visit every cell. Use **1-D** for a single sequence, **2-D** for two natural dimensions (a grid, rows × columns).




*A 2-D array (a table) with row and column indices*

### Common operations

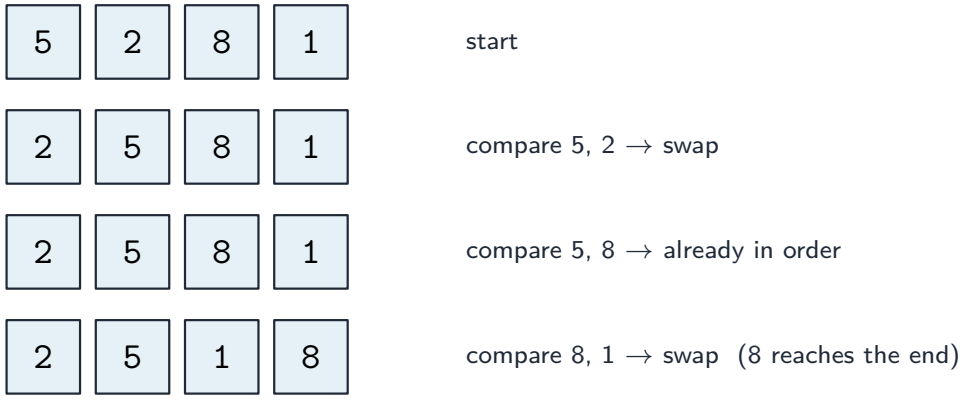
A **linear search** 线性查找 checks each element until found:

```
FOR i ← 1 TO n
    IF A[i] = Target THEN
        OUTPUT "Found at ", i
    ENDIF
NEXT i
```

To find a sum, count, maximum or minimum, set a running variable then sweep through:

```
Max ← A[1]
FOR i ← 2 TO n
    IF A[i] > Max THEN Max ← A[i]
NEXT i
```

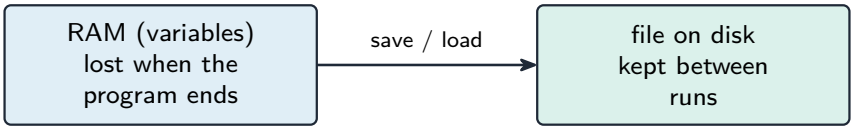
A **bubble sort** 冒泡排序 puts an array in order: pass through it comparing each adjacent pair and **swapping** any that are out of order; repeat the passes until one pass makes no swaps.




*One pass of a bubble sort: adjacent pairs are compared and swapped, bubbling the largest value to the end*

## Files

A **file** 文件 is data stored on **secondary storage** 辅助存储器, kept between program runs. Variables in RAM disappear when the program ends, so to save data permanently (high scores, records, settings) the program writes to a file. Files also let programs share data and restart from a saved state.




*Variables in RAM vanish when the program ends; a file on disk persists between runs*

A **text file** 文本文件 holds one or more lines of readable characters; programs read and write **text files** line by line. Open a file before use and **close** it after:

```
OPENFILE "data.txt" FOR READ      // or FOR WRITE, FOR APPEND
WHILE NOT EOF("data.txt") DO
    READFILE "data.txt", LineString
    OUTPUT LineString
ENDWHILE
CLOSEFILE "data.txt"
```

`EOF` tests the **end of file** 文件结束 before reading. To write:

```
OPENFILE "log.txt" FOR WRITE
FOR i ← 1 TO 100
    WRITEFILE "log.txt", "Event " & i
NEXT i
CLOSEFILE "log.txt"
```

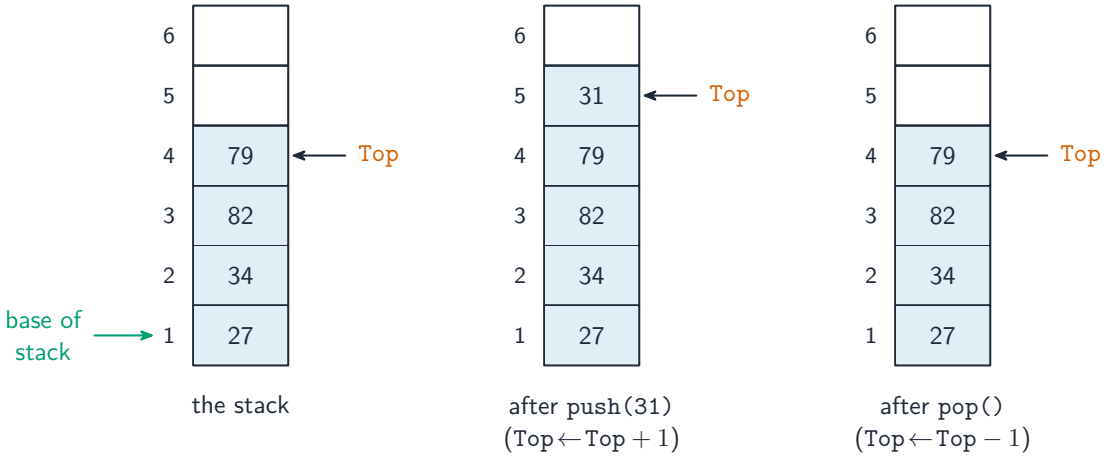
Always **close** every file — otherwise buffered writes may be lost and other programs may be locked out.

# Abstract Data Types (ADTs)

An **Abstract Data Type** 抽象数据类型 (ADT) is a **collection of data plus operations on it**, defined by **what** it does, not how it is stored. The user works only through the operations; the implementation is hidden, so it can change without affecting code that uses the ADT. Know three: stack, queue, linked list.

## Stack

A **stack** 栈 works in **LIFO** 后进先出 order (Last In, First Out). Operations: **push** 入栈 (add to the top), **pop** 出栈 (remove from the top), peek (look at the top), and tests for empty/full. Uses: undo history, function-call return addresses, expression parsing, backtracking.

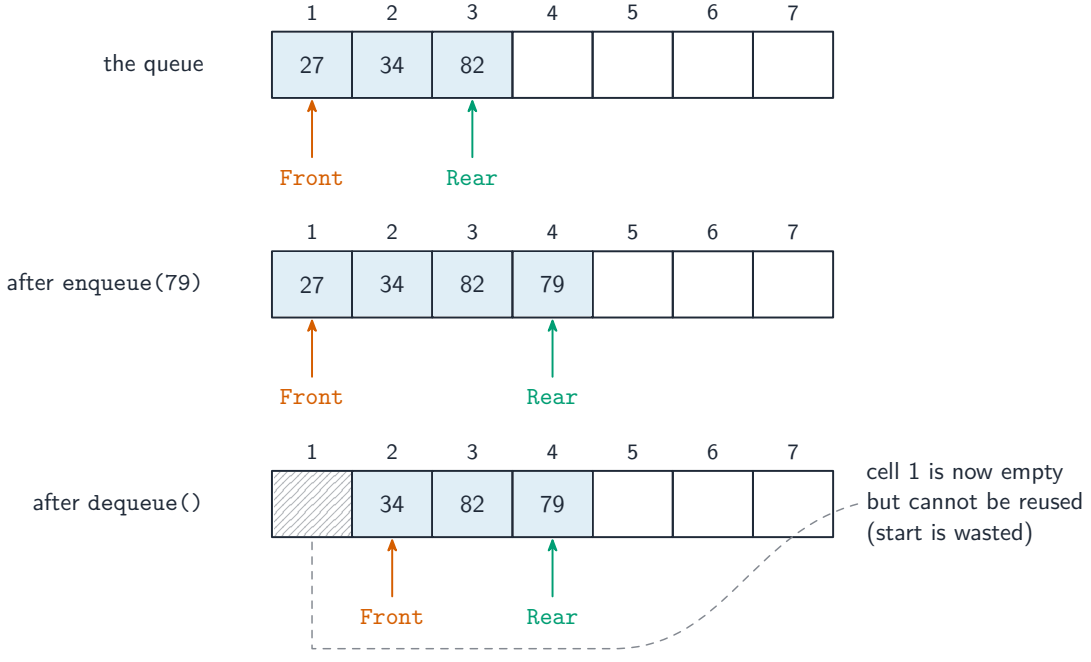



*Push and pop change the top pointer; the base pointer stays put*

*A pile of books is a stack you can see. You can only add or take a book from the **top**, so the last one you put on is the first one you take off — that is exactly LIFO*

## Queue

A **queue** 队列 works in **FIFO** 先进先出 order (First In, First Out). Operations: **enqueue** 入队 (add to the rear), **dequeue** 出队 (remove from the front), and tests for empty/full. Uses: print spooling, scheduling, breadth-first search, buffering.

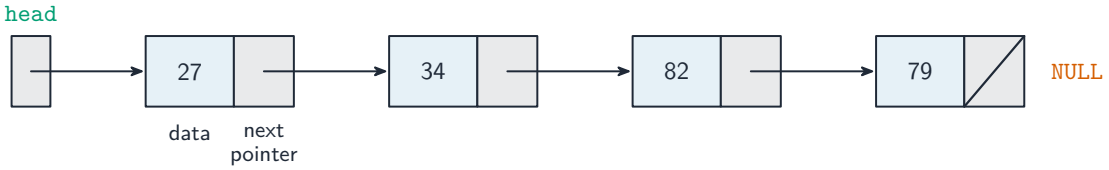



*Enqueue adds at the rear; dequeue removes from the front*

*A line of people is a queue you can see. You join at the **back** and are served from the **front**, so whoever waited longest is served first — that is exactly FIFO*

## Linked list

A **linked list** 链表 stores data as a sequence of **nodes** 节点. Each node holds a value and a **pointer** 指针 to the next node; a head pointer marks the start, and the last node's pointer is a sentinel (e.g. `NULL`). Operations: insert, delete, search, and **traverse** 遍历 (visit each node in order). Its advantage over an array is cheap insertion/deletion (just adjust pointers); its disadvantage is slow random access (you must follow pointers from the head).




*A linked list: each node points to the next*

# Implementing ADTs using arrays

## Stack using an array

Hold items in `Stack[1:MaxSize]` with an integer `Top` (0 when empty).

- `Push(x)`: if `Top = MaxSize` the stack is full (**overflow** 溢出); else `Top ← Top + 1`; `Stack[Top] ← x`.
- `Pop()`: if `Top = 0` the stack is empty (**underflow** 下溢); else return `Stack[Top]` and `Top ← Top - 1`.

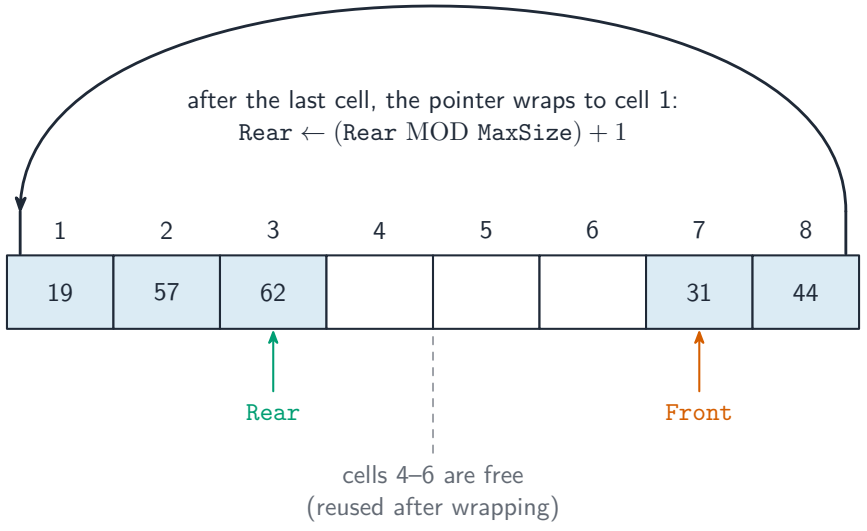
## Queue using a circular array

A simple queue lets `Front` and `Rear` march off the end, wasting the start. The fix is a **circular array** 循环数组 — when a pointer reaches `MaxSize` it wraps back to 1:

- `Enqueue(x)`: check full; else `Rear ← (Rear MOD MaxSize) + 1`; `Queue[Rear] ← x`.
- `Dequeue()`: check empty; else return `Queue[Front]` and `Front ← (Front MOD MaxSize) + 1`.

Track a separate count to tell empty from full.

For example, with `MaxSize = 6`: if `Rear = 5`, then `(5 MOD 6) + 1 = 6`, so the next item goes in cell 6; if `Rear = 6`, then `(6 MOD 6) + 1 = 1`, so the pointer **wraps back** to cell 1.




*A circular queue wraps the pointers back to the start of the array*

## Linked list using an array

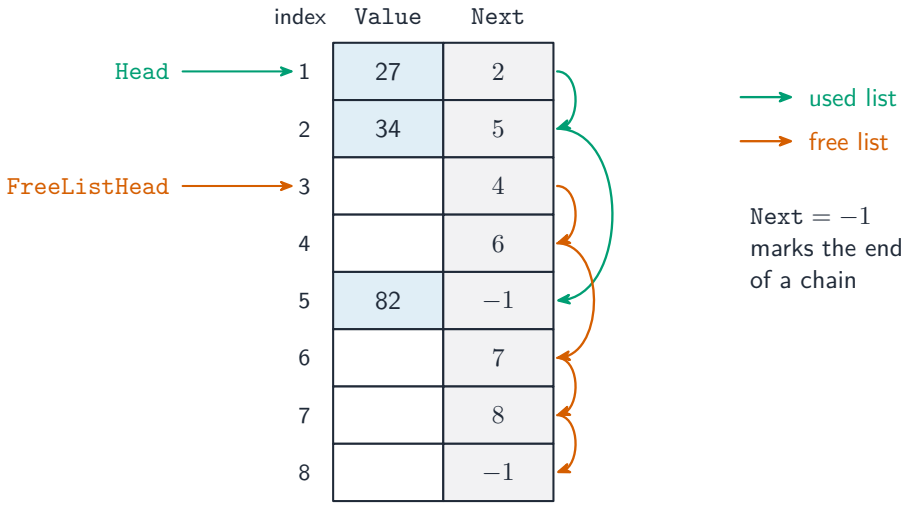
Use an array of records, each with a `Next` index:

```
TYPE TNode
    DECLARE Value : INTEGER
    DECLARE Next : INTEGER     // index of the next node, or -1 for end
ENDTYPE

DECLARE Nodes : ARRAY[1:MaxSize] OF TNode
```

```
DECLARE Head : INTEGER         // index of first node, -1 if empty
DECLARE FreeListHead : INTEGER // first available free node
```

A **free list** 空闲列表 chains the unused slots, just as the data list chains its used ones. To insert: take a slot from `FreeListHead`, set the new node's value and `Next`, and update the previous node's `Next` (or `Head`). To delete: unlink the node and return its slot to the free list. This gives the flexibility of a linked structure with the static allocation of an array.




*A linked list stored in an array: a data array and a pointer array*

**Worked example.** A circular queue is held in an array of size 5 (indices 0 to 4) with `Front = 3`, `Rear = 3` and one item stored. Two items are added, then two are removed. Where are the pointers, and why use a *circular* queue at all? Every move uses `(pointer + 1) MOD size`, so the pointers **wrap**. Adding twice moves `Rear`: $3 \to 4$, then $4 \to 0$ (because $(4+1) \bmod 5 = 0$), so `Rear = 0` and three items are stored. Removing twice moves `Front` the same way: $3 \to 4$, then $4 \to 0$, leaving `Front = 0` and one item. The wrap is the whole point: in a **linear** array queue the pointers march to the end and the freed space at the front is wasted even when the queue is empty. Remember a queue removes at the **Front** and adds at the **Rear** - a stack uses one pointer for both.

# Exam tips

- Choose the right **data structure** and justify it (a record for mixed fields, a 2-D array for a grid).
- Know how to implement a **stack, queue and linked list** with an array and pointers (top; front/rear; next).

- Distinguish an **ADT** (its behaviour) from its implementation (array plus pointers).

# Programming

A-Level Computer Science

## Programming basics

```
1 usage
public Hotel (Integer capacity) { this.capacity = capacity; }
1 usage
public synchronized boolean checkIn(AccommodationRequest request) {
    if (guests.size() < capacity) {
        guests.add(request);
        return true;
    }
    else {
        return false;
    }
}
```

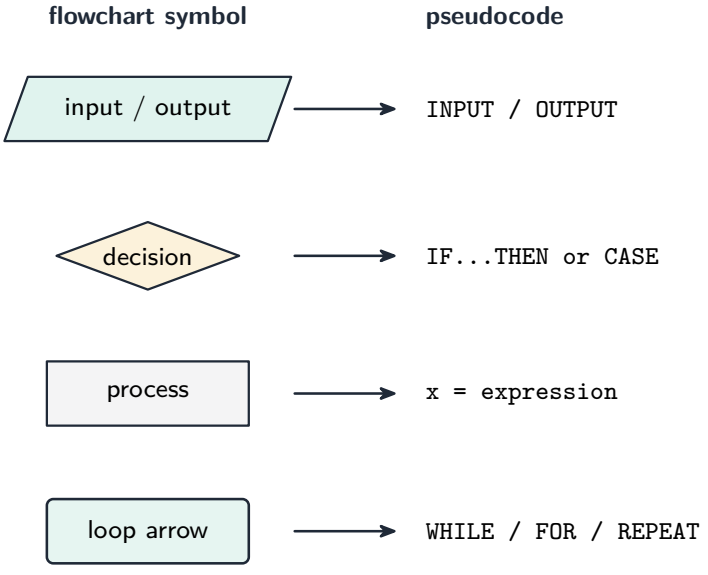
*Programming turns a design into instructions written as code*

*A programmer writes the code and tests it as they go*

## From design to code

You should be able to turn a design — a **flowchart** 流程图 (**program flowchart**) or **structured English** 结构化英语 — into **pseudocode** 伪代码, and then into a real language:

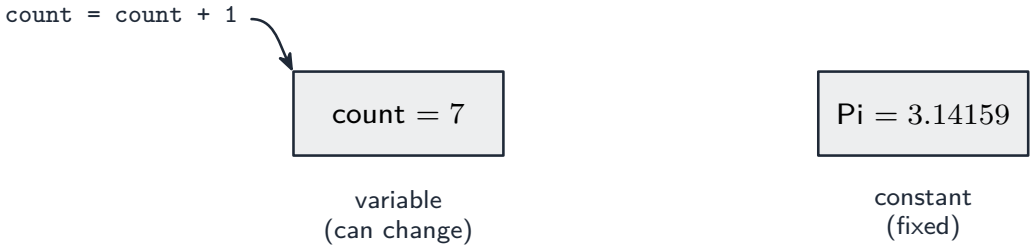
1. find the **variables** 变量 and their **data types** 数据类型.
2. turn input/output boxes into `INPUT` / `OUTPUT`.
3. turn decision diamonds into `IF...ELSE...ENDIF` (or `CASE`).
4. turn loop arrows into `WHILE`, `REPEAT...UNTIL`, or `FOR`.
5. turn process boxes into assignments or calculations.
6. check by tracing a small input.




*Each flowchart symbol becomes a pseudocode keyword*

## Constants and variables

A **constant** 常量 holds a value that **never changes**; a variable holds one that may change. Declare them with a type:




*A variable's value can change; a constant stays fixed*

```
CONSTANT Pi ← 3.14159
DECLARE Radius : REAL
DECLARE Area : REAL

Radius ← 5
Area ← Pi * Radius * Radius
```

Use constants for fixed values that recur (`Pi`, `MaxScore`); they make code clearer and easy to change in one place.

## Assignment and expressions

Use `←` for **assignment** 赋值:

```
Total ← Total + 1
Average ← Sum / Count
```

Expressions use **operators** 运算符:

- arithmetic `+ - * /`, plus `DIV` (integer division) and `MOD` (remainder): `7 DIV 2 = 3`; `7 MOD 2 = 1`.
- comparisons `=`, `<>`, `<`, `>`, `<=`, `>=`.
- logic `AND`, `OR`, `NOT`.

**Precedence** 优先级 (highest to lowest): `NOT` → `* / DIV MOD` → `+ -` → comparisons → `AND` → `OR`. Use brackets when unsure.

## Input and output

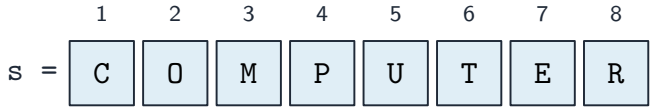
```
OUTPUT "Enter your name:"
INPUT Name
OUTPUT "Hello, ", Name
```

## Built-in functions and library routines

Many tasks have ready-made **library routines** 库例程, so you need not write them:

- string: `LENGTH(s)`, `LEFT(s, n)`, `RIGHT(s, n)`, `MID(s, start, len)`, `UCASE(s)`, `LCASE(s)`.
- numeric: `INT(x)`, `ROUND(x)`, `ABS(x)`, `MOD(a, b)`, `RANDOM()`.
- conversion: `STR(x)` (number → string), `VAL(s)` (string → number).

Use the exact names from the question paper's reference list.



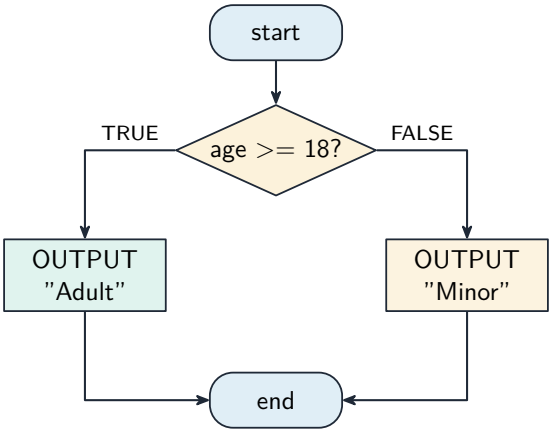

```
LENGTH(s) = 8              RIGHT(s, 2) = "ER"
LEFT(s, 3) = "COM"         UCASE(s) = "COMPUTER"
MID(s, 4, 3) = "PUT"       LCASE(s) = "computer"
```

*The common string routines acting on `s = "COMPUTER"` (positions 1–8)*

# Selection

**Selection** 选择 chooses which steps run.

```
IF age >= 18 THEN
    OUTPUT "Adult"
ELSE
    OUTPUT "Minor"
ENDIF
```

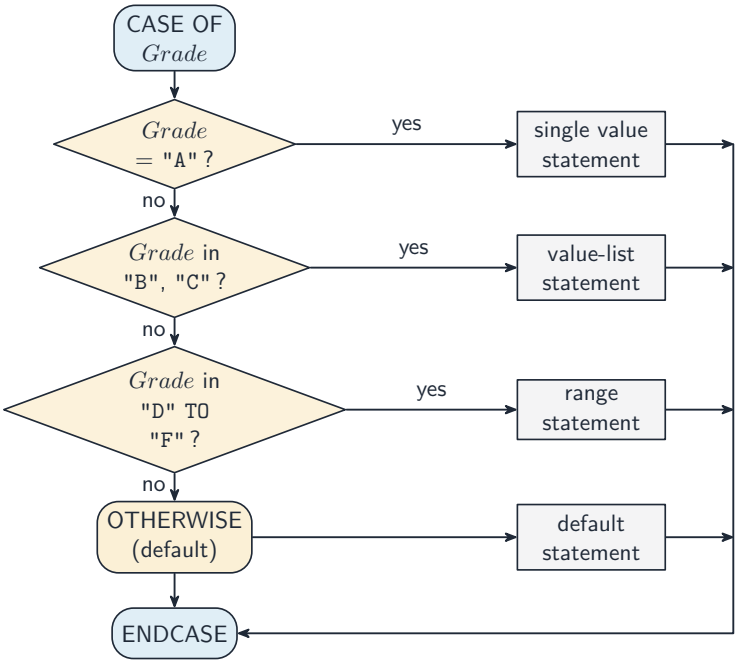



*An IF…ELSE tests the condition once, then runs exactly one branch*

For more than two cases you can use a **nested** 嵌套 IF, but deep nesting is hard to read — a `CASE` is cleaner when testing one value against several options:

```
CASE OF Grade
    "A": OUTPUT "Excellent"
    "B": OUTPUT "Good"
    OTHERWISE: OUTPUT "Try again"
ENDCASE
```

Cambridge `CASE` allows single values, value lists (`1, 2, 3:`), and ranges (`1 TO 5:`).




*A CASE statement runs the branch that matches the value*

# Iteration

**Iteration** 迭代 repeats a block. Three loops differ in how many times the body runs.

## Count-controlled (FOR) loop

A **count-controlled loop** 计数循环 — use it when you know how many times to repeat:

```
FOR i ← 1 TO 10
    OUTPUT i
NEXT i
```

A `STEP` can change the count (e.g. `FOR i ← 10 TO 1 STEP -1`). Best for a fixed number of repeats or processing each element of an **array** 数组.

## Pre-condition (WHILE) loop

A **pre-condition loop** 前测循环 tests the condition **before** each pass, so it may run **zero** times:

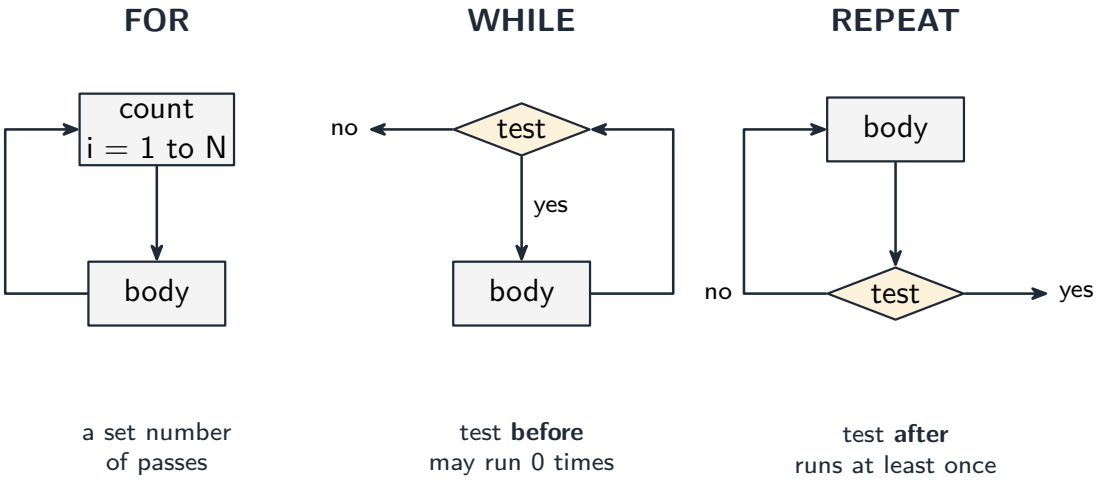
```
WHILE total < 100 DO
    INPUT n
    total ← total + n
ENDWHILE
```

## Post-condition (REPEAT...UNTIL) loop

A **post-condition loop** 后测循环 tests the condition **after** each pass, so it always runs **at least once**:

```
REPEAT
    INPUT password
UNTIL password = correctPassword
```

## Choosing the right loop




*The three loops differ in where the condition is tested — before the body (WHILE), after it (REPEAT), or a set number of times (FOR)*

- count known up front → **FOR**.
- may need zero passes → **WHILE**.
- always at least one pass → **REPEAT...UNTIL**.

Justify your choice by whether the count is known and whether the body must run at least once. A typical question gives a scenario ("ask for a password until correct, but always ask at least once") and asks which loop fits.

# Procedures and functions

**Structured programming** 结构化编程 builds a program from small named **subroutines** 子程序, each with one job.

## Procedure

A **procedure** 过程 is a named block that **does an action**; it may take **parameters** 参数 but does **not return a value**.

```
PROCEDURE Greet(name : STRING)
    OUTPUT "Hello, ", name
ENDPROCEDURE

CALL Greet("Ada")
```

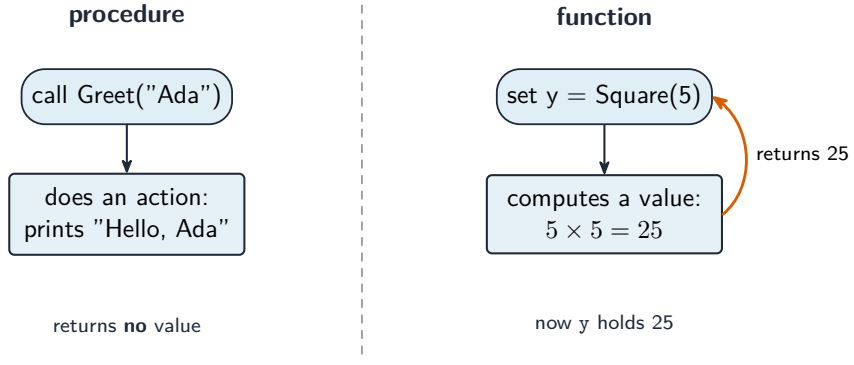
## Function

A **function** 函数 is like a procedure but it **returns a value** that becomes part of an expression.

```
FUNCTION Square(x : INTEGER) RETURNS INTEGER
    RETURN x * x
ENDFUNCTION

result ← Square(5) + 1     // result = 26
```

Use a **procedure** when the subroutine performs an action; use a **function** when it computes a value for the caller.

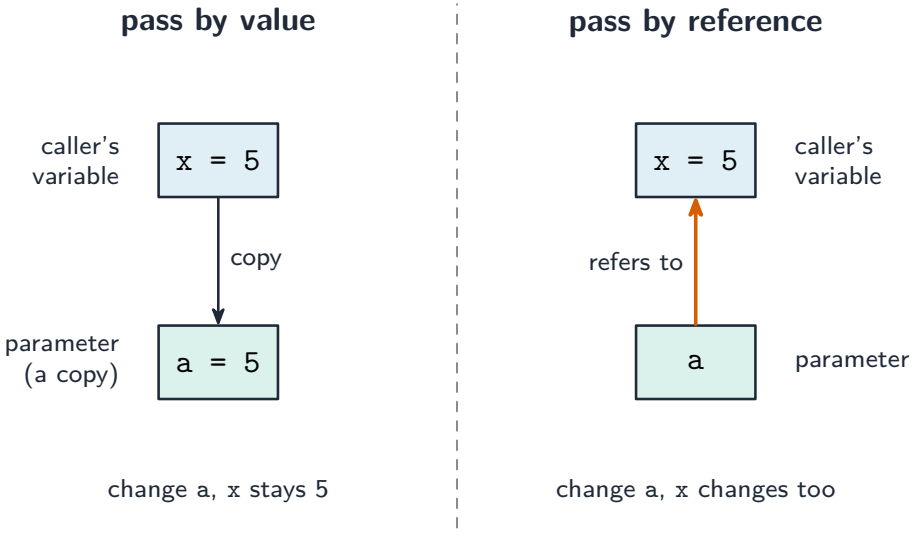



*A procedure does an action and returns nothing; a function returns a value you use in an expression*

## Parameters

A **parameter** is a variable a subroutine declares to receive input; the values the caller supplies are **arguments** 实参. Two ways to pass them:

- **pass by value** 传值 — the routine gets a **copy**; changes inside it do not affect the caller. Use for inputs it only reads.
- **pass by reference** 传引用 — the routine gets a **reference** to the caller's variable; changes **do** affect the caller. Use when it must update a parameter.

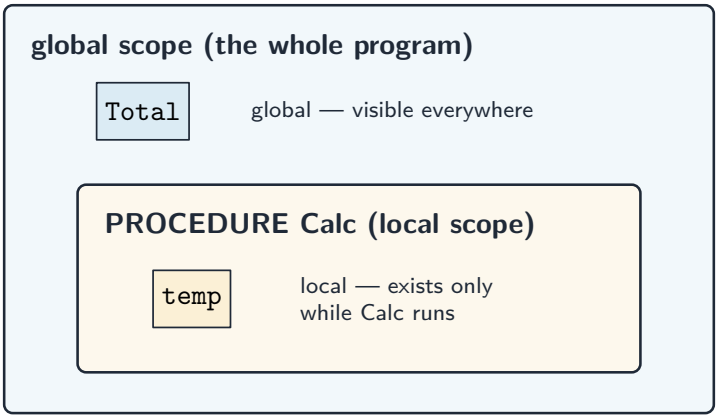



*Pass by value copies the value into a new box; pass by reference lets the routine change the caller's own variable*

```
PROCEDURE Swap(BYREF a : INTEGER, BYREF b : INTEGER)
    DECLARE temp : INTEGER
    temp ← a
    a ← b
    b ← temp
ENDPROCEDURE
```

## Local vs global variables

A **local variable** 局部变量 is declared inside a subroutine and exists only while it runs. A **global variable** 全局变量 is declared outside and is visible everywhere. Prefer locals and parameters — heavy use of globals makes code hard to follow and test. (The region where a name is visible is its **scope** 作用域.)




*A global variable is visible everywhere; a local variable exists only inside its own procedure*

## When to use a subroutine

Use a subroutine when:

- the same logic appears in **more than one place** — write it once, call it many times.
- a block has a **clear named purpose** — the name documents what it does.
- the program is **complex** — break it into parts (**decomposition** 分解).
- you want to **test a piece in isolation**.

Don't make them so tiny that the call costs more than the work inside.

## Terminology

- **definition** — the `PROCEDURE ... ENDPROCEDURE` (or function) block.
- **call** — where it is invoked. **argument** — a value passed in. **parameter** — the variable that receives it.
- **return value** — what a function passes back.
- **procedure/function header** — the first line giving the name and parameters (`PROCEDURE Name(params)` or `FUNCTION Name(params) RETURNS type`).

- **procedure/function interface** / **signature** 签名 — name + parameters + return type: what a caller must know to use it.

**Worked example.** Which loop suits each task? (a) print the 12 times table; (b) keep reading numbers until the user enters 0; (c) ask for a password until it is correct. Choose by asking **how many times** the body runs and **when the test happens**. (a) The count is known in advance (12), so use a **FOR** loop. (b) The count is unknown, and the very first input might already be 0 - so the test must come **before** the body: a **WHILE** loop, which runs **zero** or more times. (c) The count is unknown, but you must always ask **at least once** before there is anything to test - so the test comes **after** the body: a **REPEAT...UNTIL**, which runs **one** or more times. The deciding question is whether the body must run at least once: WHILE may run zero times, REPEAT always runs once.

# Writing efficient pseudocode

- **move invariants out of loops** — if a value (an **invariant** 不变量) does not change with the loop counter, compute it once before the loop.
- **exit a loop early** when the answer is found (stop a **linear search** 线性查找 as soon as the target appears).
- **avoid redundant work** — store a result and reuse it instead of recomputing.
- **choose the right data structure** — an array beats many separate variables when the items belong together.
- **replace deep nested IFs with CASE** when testing one value against many.
- **comment the intent**, not the mechanics (`// validate the postcode`, not `// loop 6 times`).
- **use meaningful names** (`numberOfPupils`, not `n`) and **initialise variables** before use.

**slower**

```
FOR i = 1 TO n
 x = slow() // every pass
 use(x, i)
NEXT i
```

**faster**

```
x = slow() // once
FOR i = 1 TO n
 use(x, i)
NEXT i
```

*Move unchanging work out of the loop so it runs once*

# Exam tips

- Distinguish a **procedure** (no return value) from a **function** (returns a value); know **pass by value vs by reference**.
- Choose the right loop: **count-controlled (FOR)** when the number of repeats is known, **condition-controlled (WHILE/REPEAT)** otherwise.
- Distinguish **local vs global** variables and scope; prefer local variables in reusable modules.

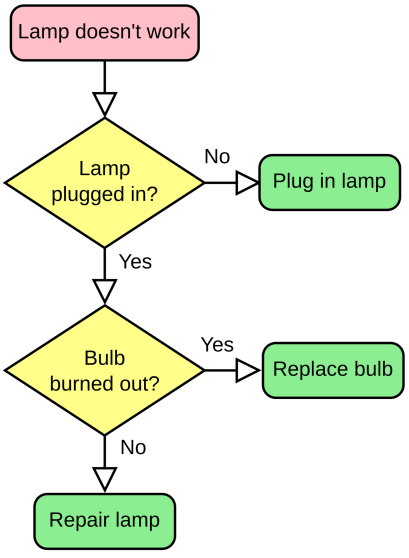
# Software Development

A-Level Computer Science

## Program development life cycle

A **development life cycle** 开发生命周期 is the set of stages from idea to finished, maintained software. It exists to **plan, manage and control** a project — to build the right product, on time, with good quality.

*Software is built by teams who follow a development life cycle to stay coordinated*




*A flowchart plans a program's logic during the design stage of the cycle*

## Why a life cycle is needed

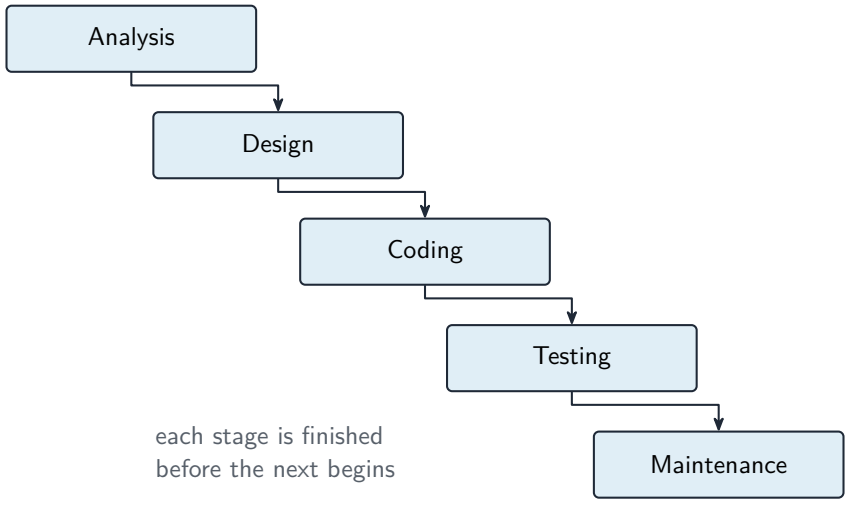
It manages complexity (break a big program into phases), coordinates teams, tracks progress with milestones, builds in testing, records design decisions for later, and manages risk.

## Why there are different ones

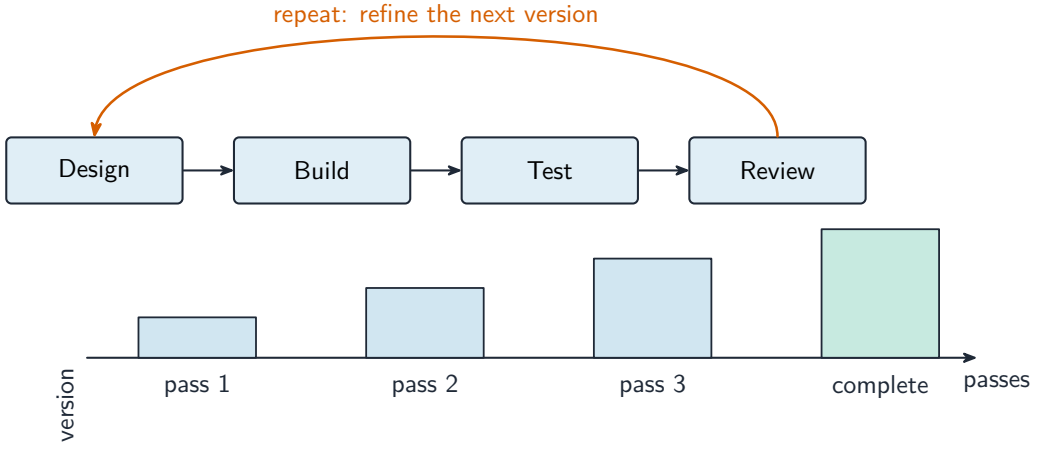
No single life cycle fits every project, so several **development life cycles** exist. The choice depends on the size and complexity, how clear the **requirements** 需求 are at the start, how much change is expected, the risk level, the team, and the deadline.

## Common models

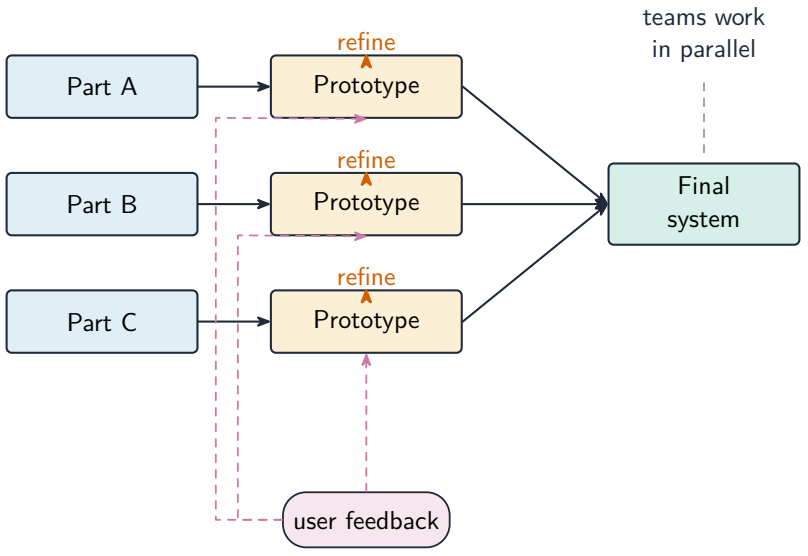
- **Waterfall** 瀑布模型 — a linear sequence (Analysis → Design → Coding → Testing → Maintenance), each stage finished before the next. Clear and well-documented; good for **stable requirements**, but poor at coping with mid-project change, and the customer sees nothing working until the end.
- **Iterative model** 迭代模型 — repeated passes, each producing a partial version that is reviewed and refined. Catches problems earlier; good when requirements are **discovered over time**, but harder to estimate.
- **Rapid Application Development** 快速应用开发 (RAD) — heavy use of a **prototype** 原型 and user feedback. Very fast first delivery; good for **changing requirements**, but depends on user availability and suits smaller systems.
- **Agile** 敏捷 — short iterations ("sprints"), constant collaboration and testing. Flexible and adaptive, but needs a committed customer and a skilled team.




*The waterfall model: each stage is finished before the next begins*




*The iterative model: repeated passes refine the program*




*Rapid application development: teams work on parts in parallel*

### The standard stages

- **analysis** — find **what** the program must do; gather and document requirements.
- **design** — decide **how**: data structures, algorithms, modules, interface, file layouts.
- **coding (implementation** 实现 **)** — write the source code following the design.
- **testing** — run against test data and fix bugs.
- **maintenance** 维护 — after release, keep it working and useful.

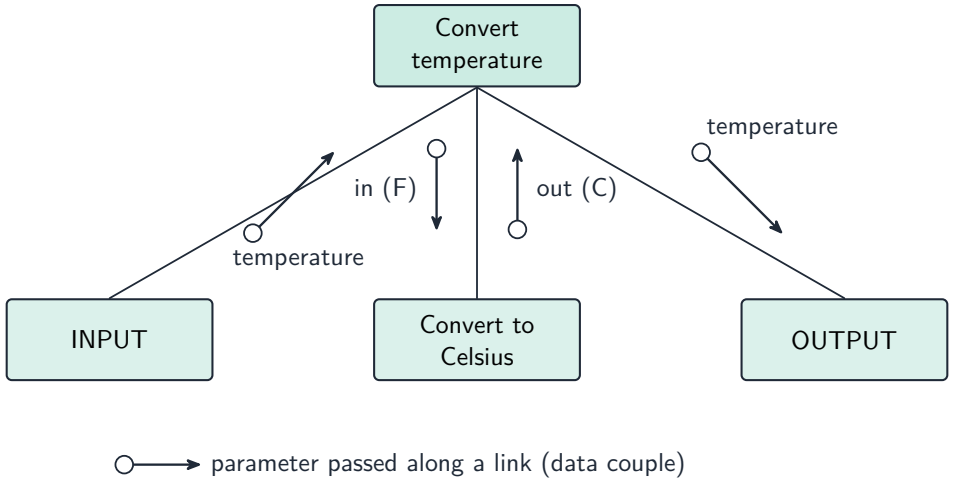
## Program design tools

### Structure chart

A **structure chart** 结构图 shows the **hierarchical decomposition** 分解 of a program into modules (**subroutines** 子程序) and the **parameters** 参数 passed between them. Each module is a rectangle; lines link caller (above) to callee (below); small arrows show data going down and results coming back up. The design can then be turned into equivalent **pseudocode** 伪代码.

```
              CalculatePay
          /        |         \
GetEmployee  CalculateBonus  CalculateTax
Returns:     Takes: sales    Takes: gross
employeeID   Returns: bonus  Returns: tax
```

It is a design-stage tool, and you can read the procedure signatures off it.



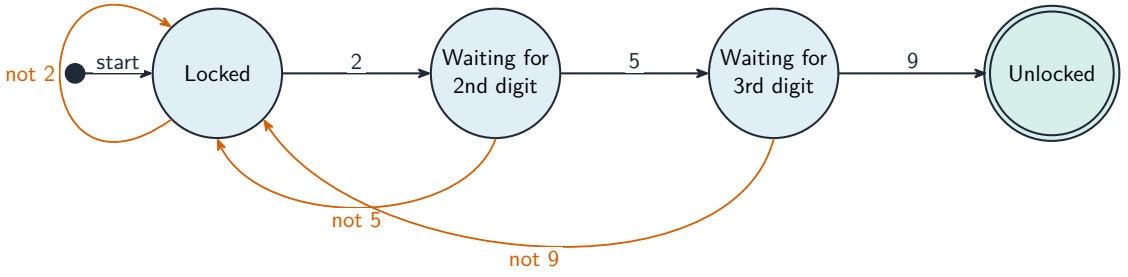

*A structure chart: modules with the parameters passed between them*

## State-transition diagram

A **state-transition diagram** 状态转换图 shows the **states** 状态 a system can be in and the **events** that move it between them — good for vending machines, traffic lights, user interfaces. **State-transition diagrams** are used to document the behaviour of an algorithm or system. Each state is a circle; each transition is an arrow labelled with the event.

```
   coin inserted                item selected
[Idle] ---------------→ [Awaiting selection] -----------→ [Dispensing]
```

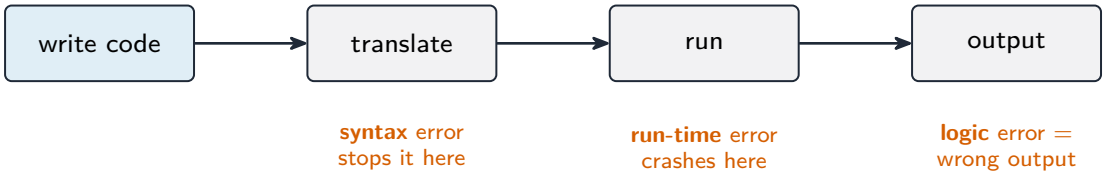
It makes missing transitions easy to spot ("what if a second coin is inserted while awaiting selection?").




*A state-transition diagram for a door lock with code 259*

# Errors

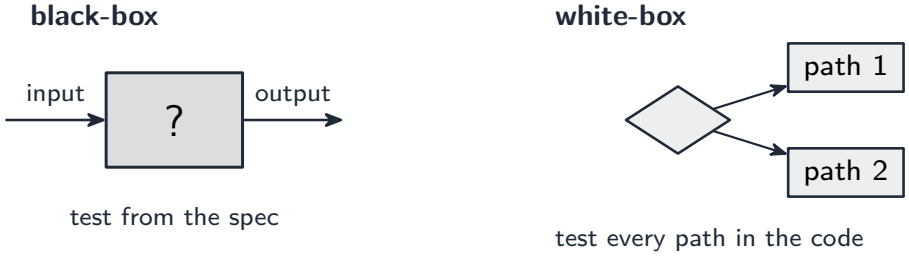
- **syntax error** 语法错误 — breaks the language's grammar (missing bracket, misspelled keyword). Caught at translation time; the program won't run until fixed.
- **run-time error** 运行时错误 — happens while running (divide by zero, file not found, array index out of range). The program crashes or raises an exception; fix by adding checks.
- **logic error** 逻辑错误 — the program runs but gives **wrong results** (using `+` for `-`, an off-by-one loop, conditions in the wrong order). The hardest to find; the only sign is wrong output, so use careful testing and tracing.




*When each error shows up: syntax at translation, run-time during the run, logic in the output*

# Testing methods

- **dry run** 手工跟踪 — trace the code on paper, writing each variable's value in a table.
- **walkthrough** 走查 — a team review of the code.
- **white-box testing** 白盒测试 — designed from the code's internal structure, covering every statement, branch and loop.
- **black-box testing** 黑盒测试 — designed from the specification only: feed inputs, check outputs.
- **integration testing** 集成测试 — combine modules and test the interfaces between them.
- **alpha testing** $\alpha$ 测试 — by the developers/in-house before release; **beta testing** $\beta$ 测试 — by a limited group of real users in their own environment.
- **acceptance testing** 验收测试 — by the customer, to decide if the product is fit for purpose.
- **stub** 桩 — a placeholder for a module that does not exist yet, so the structure can be tested top-down.




*Black-box tests the specification; white-box tests the code paths*

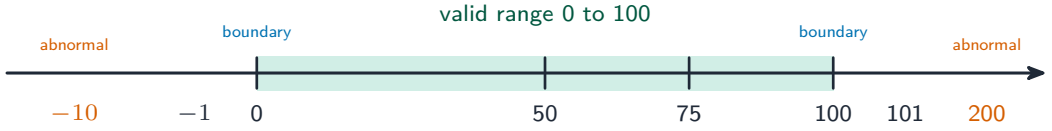
# Test strategy and test plan

A **test strategy** 测试策略 is the **high-level approach** — which kinds of testing, who does them, when, and the criteria to move on. A **test plan** 测试计划 is the **detailed list of tests** — each with input data, expected output, and a column for the actual output.

### Choosing test data

For each field or condition, include three kinds:

- **normal data** 正常数据 — typical values inside the valid range (for marks 0–100: `50`, `75`).
- **abnormal data** 异常数据 — values that should be rejected (`-10`, `200`, `"abc"`).
- **extreme data** 极端数据 — the largest and smallest values still **accepted** (`0` and `100`).
- **boundary data** 边界数据 — values at the edges, where off-by-one errors hide (each accepted extreme and the rejected value just outside it: `0`/`-1`, `100`/`101`).

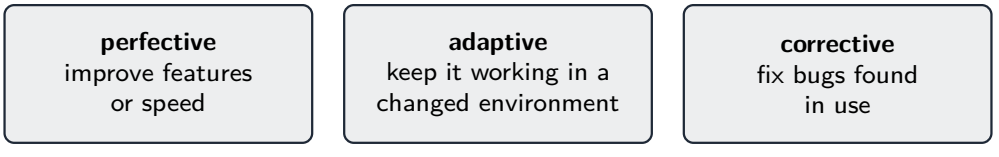



*Test data for a 0–100 field: normal inside, extremes at the boundaries, abnormal outside*

**Worked example.** A field accepts an exam mark from 0 to 100. Give test data of each kind with its expected result. **Normal**: `50` - accepted, a typical value inside the range. **Abnormal**: `-10`, `200`, `"abc"` - all rejected, being out of range or the wrong data type. **Extreme**: `0` and `100` - the largest and smallest values that are still **accepted**. **Boundary**: the pairs straddling each edge - `-1` rejected alongside `0` accepted, and `100` accepted alongside `101` rejected. Every value must carry its **expected result**, or the test plan proves nothing. Extreme and boundary are the pair most often confused: an extreme value sits **inside** and is accepted, while a boundary test is always a **pair** either side of the edge - which is exactly where off-by-one errors hide.

## Maintenance

Most of a program's lifetime cost is in maintenance. Three kinds:




*Three kinds of maintenance: perfective, adaptive and corrective*

- **perfective maintenance** 完善性维护 — improving performance or features even though it works (a faster query, a new option).
- **adaptive maintenance** 适应性维护 — keeping it working in a changing environment (a new OS, a new API, a legal change).
- **corrective maintenance** 纠正性维护 — fixing bugs found in use.

A program may need all three throughout its life.

## Amending an existing program

When asked to add a feature or fix a bug:

1. **read the existing code** until you understand the algorithm and data flow.
2. **find where the change goes** — which subroutine, which lines.
3. **make the change as small as possible** — don't rewrite working code.
4. **update related parts** — every caller of a changed parameter list, every routine using a changed data structure.
5. **test** the new behaviour **and** the old (**regression testing** 回归测试 — check you broke nothing).
6. **document** the change.

Clear comments, meaningful names, decomposed subroutines and a structure chart make a program much easier to amend — which is why the design tools matter even after the first release.

## Exam tips

- Compare development models (waterfall, iterative, RAD) and know the stages of the **program development life cycle**.
- Distinguish **syntax, logic and run-time** errors and how each is found.
- Choose **test data** of three kinds — normal, boundary and erroneous — and give an example of each for the stated range.
- Distinguish the types of **maintenance** (corrective, adaptive, perfective).

# Data Representation

A-Level Computer Science

## User-defined data types

The built-in types (`INTEGER`, `REAL`, `STRING`, `CHAR`, `BOOLEAN`) cover the simplest cases. For richer problems you can define **user-defined types** 用户定义类型, making the code clearer and the compiler stricter.

### Why they are needed

A built-in `STRING` lets you store nonsense in a field that should hold one of a few legal values; a user-defined type can restrict it. Real entities are usually a **collection** of values of different types. And `DECLARE Taxi : Vehicle` is clearer (self-documenting) than `DECLARE Taxi : STRING`.

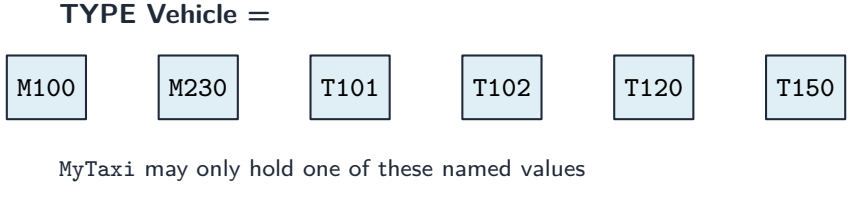
### Non-composite types

#### Enumerated type

An **enumerated type** 枚举类型 has values that are a **fixed list of named constants**:

```
TYPE Vehicle = (M100, M230, T101, T102, T120, T150)
DECLARE MyTaxi : Vehicle
MyTaxi ← T102
```

The names are values of the new type (stored internally as small integers); you cannot assign anything outside the list. Uses: days of the week, colours, status codes.



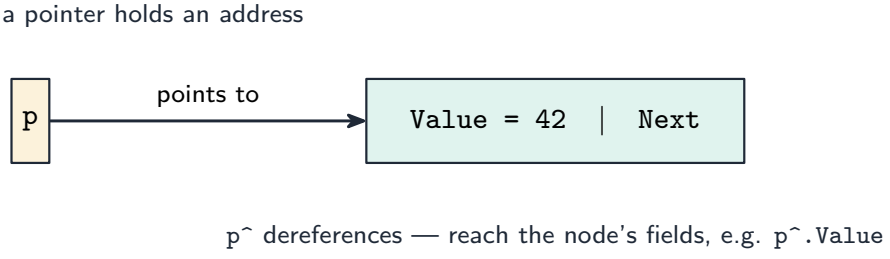

*An enumerated type is a fixed list of named values*

### Pointer type

A **pointer** 指针 holds the **memory address of another variable** (or `NULL` for "no target"). Pointers build dynamic structures (linked lists, trees) and pass references without copying.

```
TYPE PNode = ^TNode    // pointer to a TNode
DECLARE p : PNode
p ← NEW TNode
p^.Value ← 42          // dereference to reach the fields
```

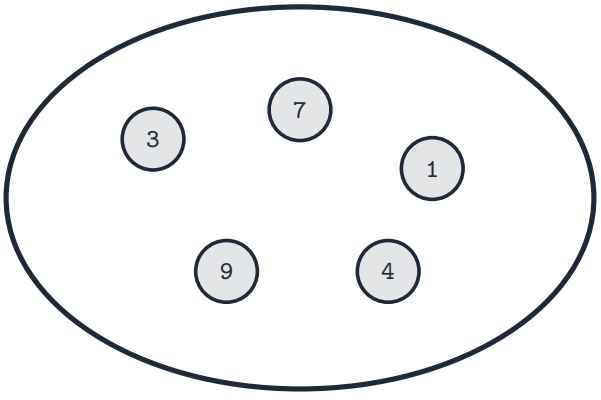
To **dereference** 解引用 (`p^`) means to reach the variable it points to.




*A pointer holds an address; `p^` dereferences it to reach the node's fields*

## Composite types

A **composite type** 复合类型 (one of the **composite data types**) groups several values under one name.




unordered, every value unique

*A set is an unordered collection of unique values*

record Student

| Name : STRING |
|---|
| Age : INTEGER |
| Grade : CHAR |
| Enrolled : BOOLEAN |

one name, several fields of different types

*A record groups fields of different types under one name*

- **record** 记录 (Topic 10) — fields of different types in a `TYPE ... ENDTYPE` block.
- **set** 集合 — an unordered collection of unique values, with operations add, remove, membership test, union, intersection:

```
DECLARE Available : SET OF Colour
Available ← {Red, Blue}
IF Green IN Available THEN ...
```

- **class** 类 / **object** 对象 — the OOP composite type, combining data fields (**attributes** 属性) with operations on them (**methods** 方法). An object is an instance of a class:

```
CLASS Taxi
    PRIVATE Capacity : INTEGER
    PUBLIC FUNCTION GetCapacity() RETURNS INTEGER
        RETURN Capacity
    ENDFUNCTION
ENDCLASS
```

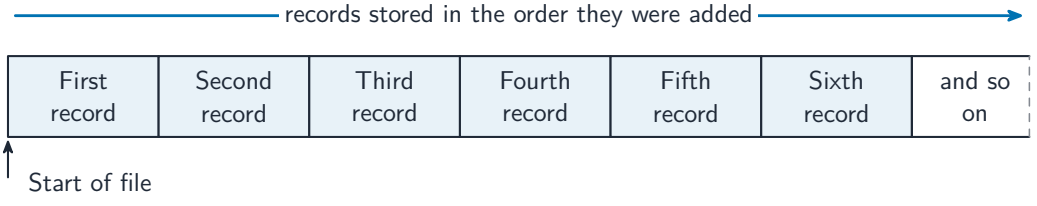
### Choosing a type

Use **enumerated** for a value from a fixed list, **pointer** for indirection, **record** for a group of fields, **set** for an unordered unique collection, and **class** when you need state **and** behaviour together.

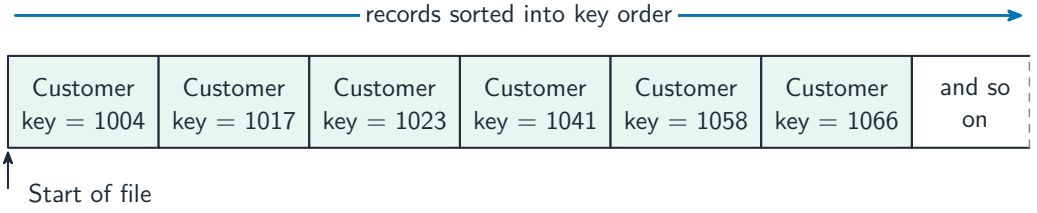
## File organisation and access

**File organisation** 文件组织 is how the data is laid out; **file access** is how the program reaches a record.

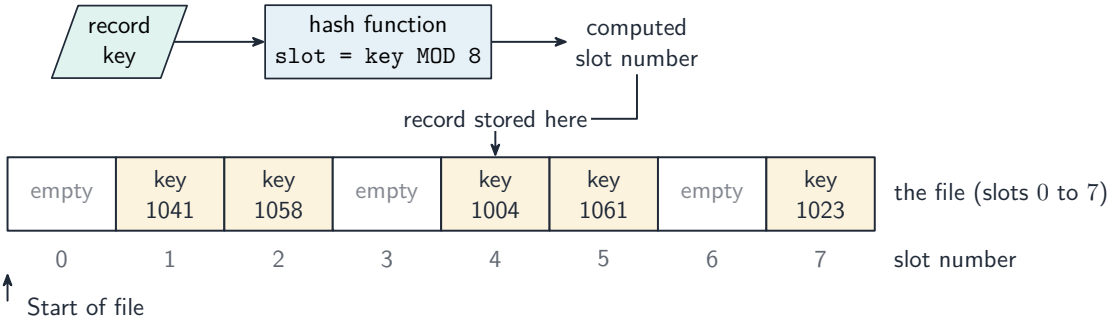
- **serial file** 串行文件 — records in the **order added**, no sorting. Access is sequential only; appending is fast; searching is slow. Used for logs and audit trails.
- **sequential file** 顺序文件 — records **sorted by a key**. Searching is faster (you can stop early or binary-search); inserting is slow (records must shift). Used for master files updated in batch.
- **random (direct-access) file** 随机文件 — records at positions computed from the key (often by a hash). Direct access by key is very fast; reading in key order is harder. Used for large lookup tables and customer accounts.




*Serial file: records are kept in the order they were added*




*Sequential file: records are sorted by a key field*




*Random file: records sit at positions computed from the key*

The two access methods are **sequential access** 顺序存取 (read from start to end) and **direct access** 直接存取 (jump straight to a known position). Match the structure to the dominant operation: single-key lookups favour random; in-order reports favour sequential.

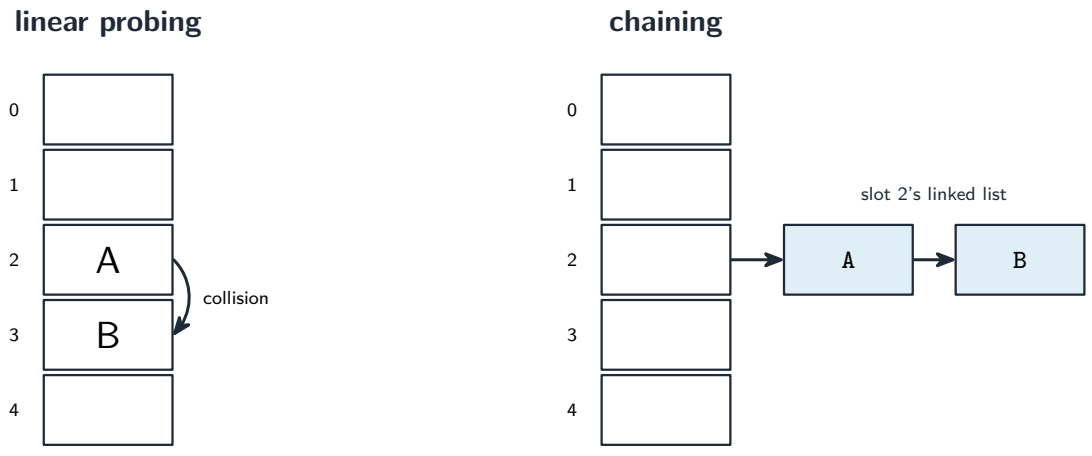
# Hashing

A **hash function** 散列函数 (a **hashing algorithm**) takes a record key and produces an **address** where the record is stored. A good one is fast, **deterministic** 确定性, and spreads keys evenly.

Common **hashing algorithms** for $N$ slots: modulo hash `address ← key MOD N`; folding (split the key, add the pieces, MOD N); a string hash (sum the character codes, MOD N).

A **collision** 冲突 is when two keys hash to the same address. Three ways to resolve it:

| Strategy | How it works | Trade-off |
|---|---|---|
| **linear probing** 线性探测 | use the next free slot (wrapping around) | simple, but keys **cluster** |
| **chaining** 链接法 | each slot points to a **linked list** 链表 of records | no clustering, but uses more memory |
| rehashing | apply a second hash function | spreads keys, but more work |




*Resolving a hash collision: linear probing uses the next free slot; chaining keeps a linked list per slot*

To search: hash the key, read that slot; if the keys match you are done, else follow the resolution strategy until a match or an empty slot. To insert: hash the key, write to that slot or the next free one. Keep the **load factor** 装填因子 (records ÷ slots) below about 70% for near-O(1) lookups.

# Floating-point numbers

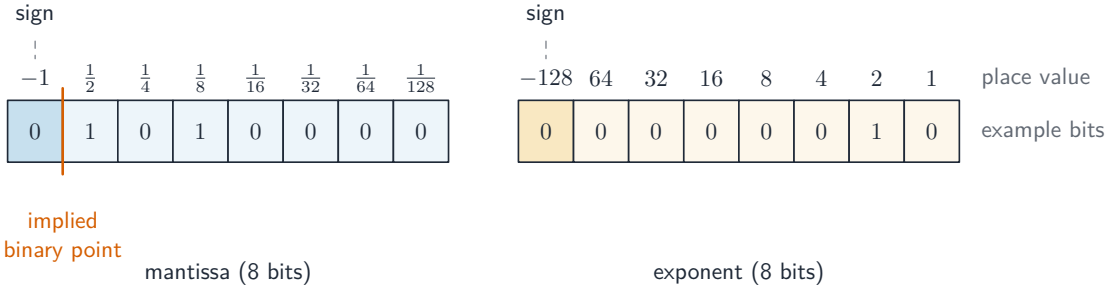
To store **real numbers** of very different sizes, computers use a **floating-point** 浮点 format — a binary form of scientific notation, with two fields:

- a **mantissa** 尾数 — the significant digits.
- an **exponent** 指数 — the power of 2 to multiply by.

Both are stored as **two's complement** 补码 integers. The value is

$$\text{number} = \text{mantissa} \times 2^{\text{exponent}}.$$

Read the mantissa as a binary fraction — the first bit after the point is worth 1/2, the next 1/4, then 1/8, and so on. So `0.1010000` is $1/2 + 1/8 = 0.625$; with exponent `00000010` ($= 2$) the value is $0.625 \times 2^2 = 2.5$.




*The place values of an 8-bit mantissa and an 8-bit exponent*

## Converting

- **binary → denary**: read the mantissa (use two's-complement rules if negative) as a fraction, read the exponent as a signed integer, then multiply mantissa by $2^{\text{exponent}}$.
- **denary → binary**: write the number as a binary fraction × a power of 2, then store the mantissa and exponent in the agreed formats.

**Worked example.** A number has mantissa `10110000` and exponent `00000011`. Find its denary value.

The exponent `00000011` is +3. The mantissa begins with a 1, so it is negative. Read as `1.0110000` in two's complement, the sign bit is worth −1 and the fraction bits add $\frac{1}{4} + \frac{1}{8} = 0.375$, so the mantissa is $-1 + 0.375 = -0.625$. Then

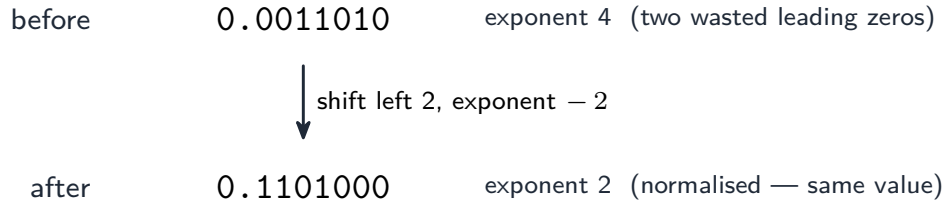
$$\text{number} = -0.625 \times 2^3 = -5.0.$$

**Worked example.** Store +2.5 in this format.

In binary $2.5 = 10.1$. Written as a normalised fraction, $2.5 = 0.101 \times 2^2$. So the mantissa is `01010000` (sign bit 0, then `.101`) and the exponent is `00000010` (= 2).

## Normalisation

A number is **normalised** 规格化 when the first significant bit is **immediately after the binary point** (no wasted leading zeros). This **maximises precision**, because every mantissa bit carries information. To normalise, shift the mantissa left and decrease the exponent (or shift right and increase it) until the first significant bit is in place; the value is unchanged. For negative (two's-complement) mantissas, the sign bit (1) is followed immediately by a 0.




*Normalising: shift the mantissa left to remove leading zeros, lowering the exponent by the same amount*

## Approximation and rounding errors

Many denary reals **cannot be stored exactly** in binary — e.g. $0.1_{10}$ is the repeating binary fraction $0.000110011\ldots_2$, which must be truncated. Consequences:

- **rounding errors** 舍入误差 build up over many operations (`0.1 + 0.2` is not exactly `0.3`).
- **comparisons fail** — test `ABS(x - 0.3) < 1e-9` instead of `x = 0.3`.
- subtracting two nearly-equal values loses precision.
- **overflow** 溢出 (a result too large for the exponent's range) and **underflow** 下溢 (a result too small, rounding to zero) occur when the exponent runs out of range.

For exact needs (currency), use **fixed-point** 定点 or **BCD** 二进码十进数 instead of floating-point.

# Exam tips

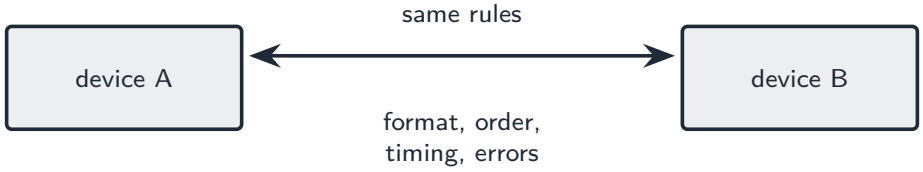
- For **floating-point** binary → denary, read the mantissa as a fraction and the exponent as a signed integer, then multiply; a **negative mantissa** follows two's-complement rules.
- **Normalise** by shifting until the first digit after the point differs from the sign bit — this maximises precision.
- Explain **rounding, overflow and underflow** errors and why 0.1 cannot be stored exactly.
- Compare file organisation (serial, sequential, direct) and how **hashing** finds a record quickly.

# Communication and internet technologies

A-Level Computer Science

## Why protocols are needed

A **protocol** 协议 is a **set of rules** for how devices **communicate**. Both ends must follow the same rules, or one side's signals are meaningless to the other. Protocols define the **format** of the data (where addresses and payload sit), the **order** of messages (who speaks first, when to acknowledge), the **meaning** of each message, the **timing** (timeouts, retransmits), and **what to do on error**. Without an agreed protocol, communication fails — like two people speaking different languages with no translator.



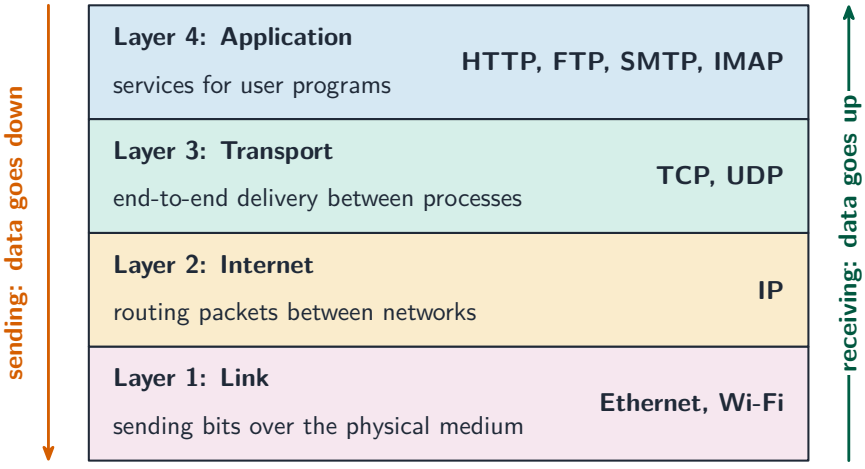

*A protocol is the shared rules: format, order, timing and errors*

## Layered protocols

Networking is complex, so it is split into **layers** 层, each with one focused job, talking only to the layer above and below. Benefits: **modularity** 模块化 (replace one layer — say Ethernet with Wi-Fi — without touching the others), standardisation (vendors interoperate), and **abstraction** 抽象 (you ignore details handled elsewhere). The internet uses the **TCP/IP protocol suite** 协议栈 (4 layers).

## TCP/IP protocol suite

| Layer | Purpose | Examples |
|---|---|---|
| **Application** | what the user program does | HTTP, FTP, SMTP, IMAP |
| **Transport** | end-to-end delivery between processes | TCP, UDP |
| **Internet** | routing packets between networks | IP |
| **Link** | sending bits over the physical medium | Ethernet, Wi-Fi |



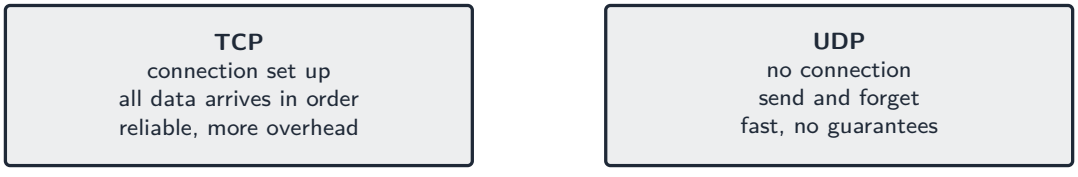

*The four layers of the TCP/IP protocol suite*

## Application layer

The **application layer** 应用层 gives services to user programs and defines the protocols they speak (HTTP for web, SMTP for email). This is where a programmer most often works.

## Transport layer

The **transport layer** 传输层 delivers data **end-to-end** between processes, identified by **port numbers** 端口号. Two protocols:




*TCP connects and delivers in order; UDP sends and forgets*

- **TCP** 传输控制协议 — **connection-oriented** 面向连接: sets up a connection, ensures **all data arrives in order**, retransmits lost **packets** 数据包, controls flow. Reliable but with overhead. Used by HTTP, HTTPS, SMTP, FTP.
- **UDP** 用户数据报协议 — **connectionless** 无连接: sends and forgets, with no acknowledgements or ordering. Low overhead, no guarantees. Used for streaming, DNS and gaming, where speed beats reliability.

## Internet layer

The **internet layer** 网络层 carries packets between hosts using **IP**. Each packet has a source and destination **IP address** IP 地址, and **routers** 路由器 forward it onward. It does not guarantee delivery — that is TCP's job.

A home router does this job for your house: it reads each packet's destination address and sends it on towards the internet, and back to the right device.

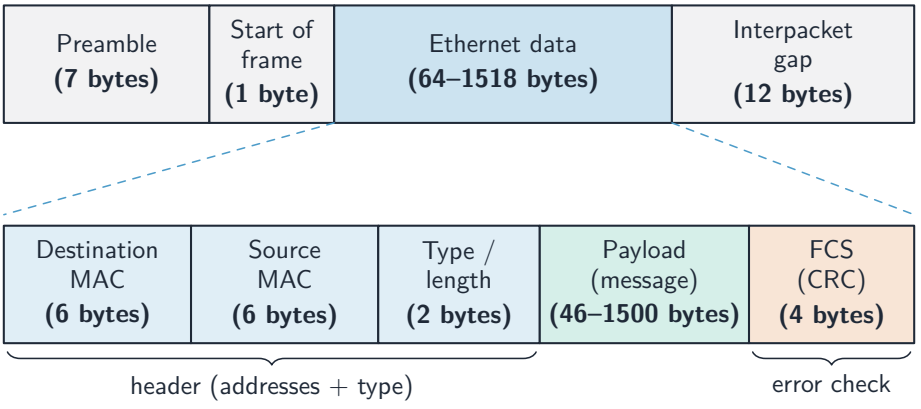
*A home Wi-Fi router: it forwards packets between your devices and the internet*

Before the router reaches the wider internet, a **modem** 调制解调器 connects the home to the internet provider over the provider's cable or phone line. Its lights show the link is up and online.


*A cable modem connects a home network to the internet provider*

## Link layer

The **link layer** 链路层 sends bits over one physical link (Ethernet, Wi-Fi). It adds a frame header with **MAC addresses** MAC 地址 and handles medium access (e.g. **CSMA/CD** 载波侦听多路访问/冲突检测 on Ethernet).

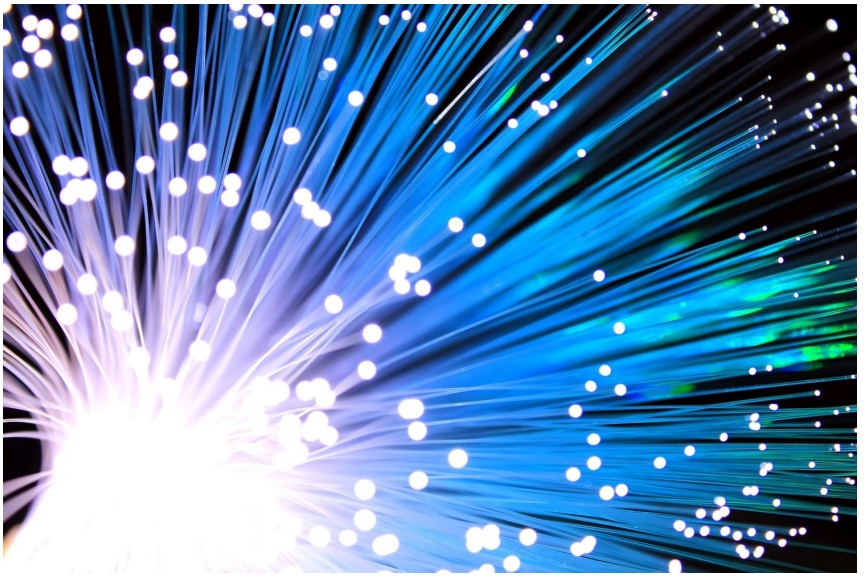



*The parts of a typical Ethernet frame*

On a wired local network, a **switch** 交换机 joins many devices together. Each device plugs into a port with an Ethernet cable (an RJ45 plug), and the switch uses the MAC addresses in each frame to send it only to the correct port.


*A network switch connects many wired devices on a local network*

The physical link can be a copper wire, a radio signal (Wi-Fi), or a **fibre-optic cable** 光纤. In a fibre-optic cable, the bits travel as flashes of light through very thin strands of glass, which is fast and carries data a long way.



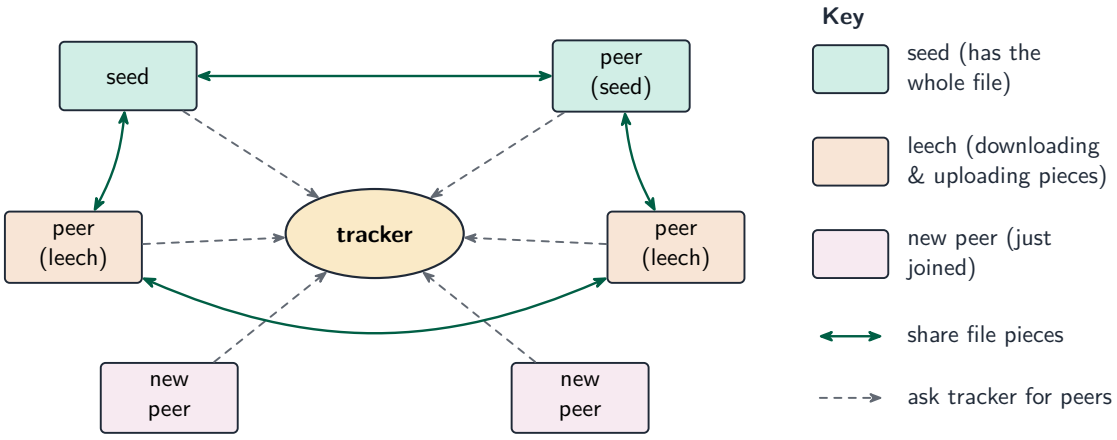
*A fibre-optic cable: data travels as light through thin glass strands*

A radio link can reach much further. A **satellite dish** 卫星天线 sends and receives radio signals to and from a satellite, carrying data to places that wired links cannot easily reach.

*A satellite dish sends and receives data by radio over a long distance*

## Common application-layer protocols

- **HTTP** 超文本传输协议 — browsers fetch web pages from servers (over TCP, port 80). **HTTPS** is HTTP over TLS — encrypted, port 443.
- **FTP** 文件传输协议 — transfer files between client and server.
- **SMTP** 简单邮件传输协议 — **send** email between client and server, and between servers. Receiving uses POP3 or IMAP.
- **POP3** — downloads email and usually deletes it from the server. **IMAP** — leaves email on the server and syncs across devices, so the same inbox appears everywhere.
- **BitTorrent** — a **peer-to-peer** 对等网络 protocol; a file is split into pieces downloaded from many peers in parallel, so no single server carries all the load.

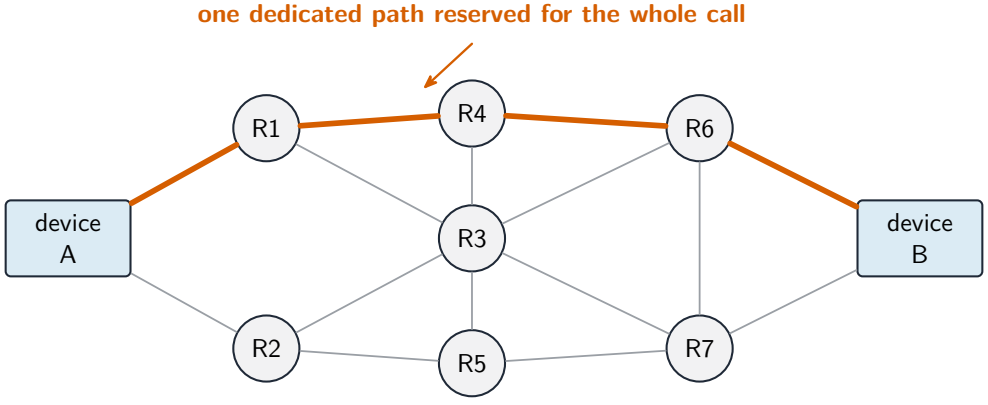



*BitTorrent: a tracker helps peers find each other, then they share file pieces directly*

# Circuit switching vs packet switching

## Circuit switching

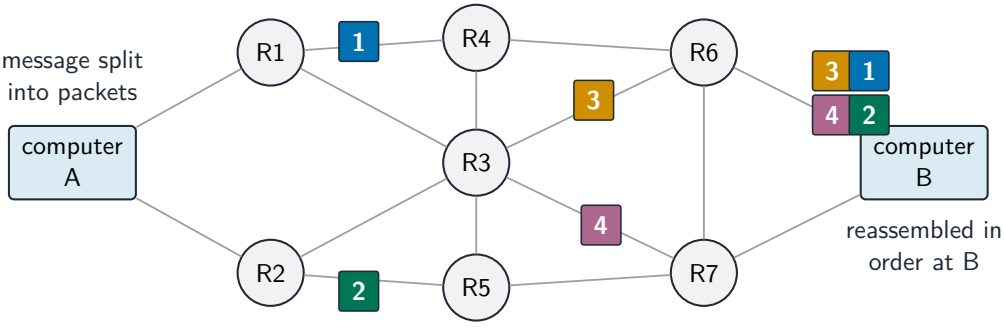
A **dedicated path** is set up between the two ends before any data is sent (**circuit switching** 电路交换), reserved for the whole conversation, then released. It gives **reserved bandwidth** 带宽 and in-order delivery, but is **inefficient** during silences and slow to set up. Classic example: the traditional telephone network.




*Circuit switching: one dedicated path is reserved end to end*

## Packet switching

The data is split into **packets**, each sent **independently** (**packet switching** 分组交换). Each packet carries the destination address; routers make per-packet decisions, so packets may take different routes and arrive out of order, and the destination reassembles them. It is **efficient** (one link is **multiplexed** 多路复用 across many conversations), **robust** (reroute around a failure), but has **variable latency** 延迟 and possible loss (TCP handles reliability). Used by the internet.




*Packet switching: packets travel independently and may take different routes*

| Aspect | Circuit switching | Packet switching |
|---|---|---|
| Path | dedicated, reserved | shared, per-packet |
| Setup time | slow | none |
| Bandwidth use | inefficient | efficient |
| Order | in order | may be out of order |
| Robustness | one failure cuts the circuit | reroute around failures |
| Suits | constant-rate flows (voice) | bursty flows (web, email) |

Modern networks use packet switching for its efficiency and resilience.

## Describing packet switching in a few sentences

A good exam answer: "The message is broken into small packets. Each packet carries the destination and source addresses and a sequence number. Each packet travels through the network independently, with routers choosing the next hop per packet. Packets may take different paths and arrive out of order. The destination uses the sequence numbers to reassemble the message, and missing packets can be requested again."

**Worked example.** A phone call and a large file download share a network. Which switching method suits each, and why? A **phone call** needs a steady stream with low delay, and it would suffer badly if pieces arrived late or out of order - so **circuit switching** suits it: a dedicated path is set up for the whole call and its capacity

is reserved for the duration. A **file download** does not care about timing or arrival order, because the receiver reassembles it, and it benefits from using whatever capacity happens to be spare - so **packet switching** suits it: the file is split into packets that travel **independently**, each carrying source and destination addresses and a **sequence number**, with routers choosing a next hop per packet. Name the **property of the traffic** that decides it: reserved capacity and low delay for the call, efficiency and resilience for the download.

# Exam tips

- Explain **why protocols and layers** are used: each layer has one job and can change independently.
- Place common protocols in the **TCP/IP** stack (HTTP/FTP/SMTP application; TCP/UDP transport; IP internet).
- Compare **circuit switching vs packet switching** (a dedicated path vs independent packets) with a use for each.

# Hardware and Virtual Machines

A-Level Computer Science

## RISC vs CISC processors

Two styles of CPU design. The CPU itself plugs into the **motherboard** 主板, the main board that links the processor, the memory and every other part of the computer together.

**CISC**
many, complex instructions
variable length
does more per instruction

**RISC**
few, simple instructions
fixed length
each instruction is fast

*CISC has many complex instructions; RISC has few simple ones*


*A motherboard links the CPU, memory and other parts together*

## CISC

A **CISC** 复杂指令集 (Complex Instruction Set Computers) has **many, often complex** instructions (one may do several memory accesses and operations), of **variable length**, so decoding is intricate. It does more per instruction in hardware. Examples: Intel x86.

## RISC

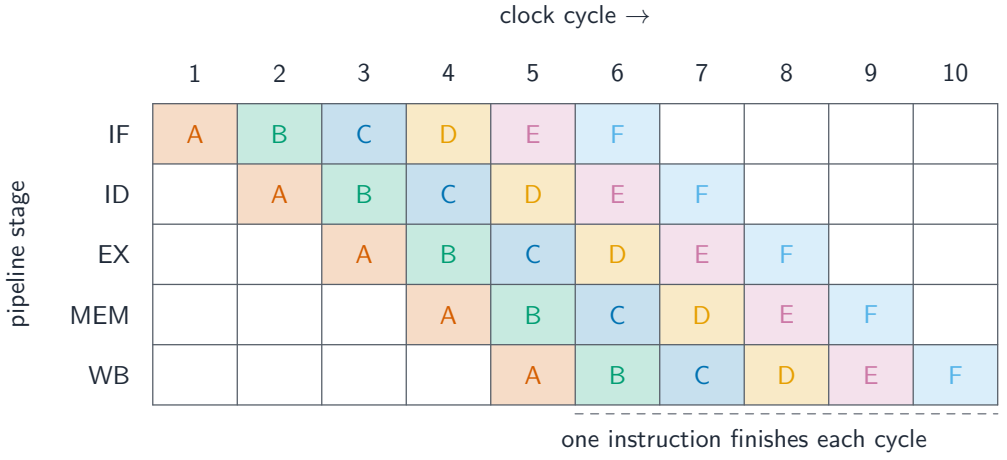
A **RISC** 精简指令集 (Reduced Instruction Set Computers) has a **small set of simple** instructions, each doing one basic operation, all of **fixed length** (fast to decode). Only **load** and **store** touch memory; everything else is **register** 寄存器-to-register. Programs are longer but each instruction is quick and predictable, which suits pipelining. Examples: ARM, RISC-V.

| **Feature** | **CISC** | **RISC** |
|---|---|---|
| Instruction set | many | few |
| Instruction length | variable | fixed |
| Memory access | many instructions | only load/store |
| Pipeline-friendly | harder | naturally |
| Per-instruction cycles | varies | usually 1 |

The trade-off is doing **more per instruction** (CISC) vs doing each instruction **faster and more predictably** (RISC). Modern Intel chips translate CISC instructions into simpler RISC-like micro-ops internally.

# Pipelining

A **pipeline** 流水线 processes instructions in **overlapping stages**, like an assembly line: Fetch → Decode → Execute (in the **ALU** 算术逻辑单元) → Memory access → Write back. Each stage works on a different instruction at once, so once the pipeline is full, **one instruction completes per cycle**. RISC's fixed-length, simple instructions make every stage take the same time. A pipeline can stall on a **hazard** 冒险 — a data hazard (an instruction needs a result not ready yet) or a control hazard (a branch makes the next address unknown).




*Pipelining overlaps the stages of six instructions, so one finishes each cycle*

RISC chips keep data in **many registers** because memory is slow and registers are fast; the compiler allocates values to registers wisely.

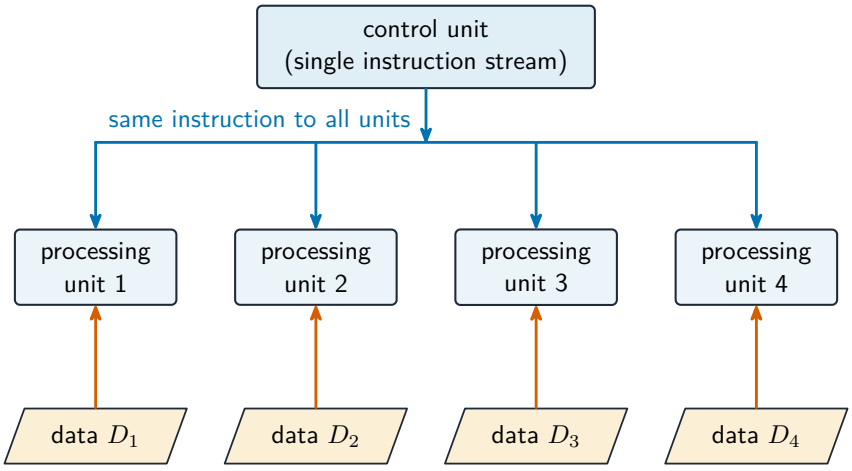
A processor running this fast gives off a lot of heat, so a **heat-sink** 散热器 and fan sit on top of it. The metal fins spread the heat and the fan blows it away, keeping the CPU cool enough to work.


*A CPU heat-sink and fan carry heat away from the processor*

## Flynn's taxonomy

**Flynn's taxonomy** 弗林分类 sorts computers by the number of instruction and data streams:

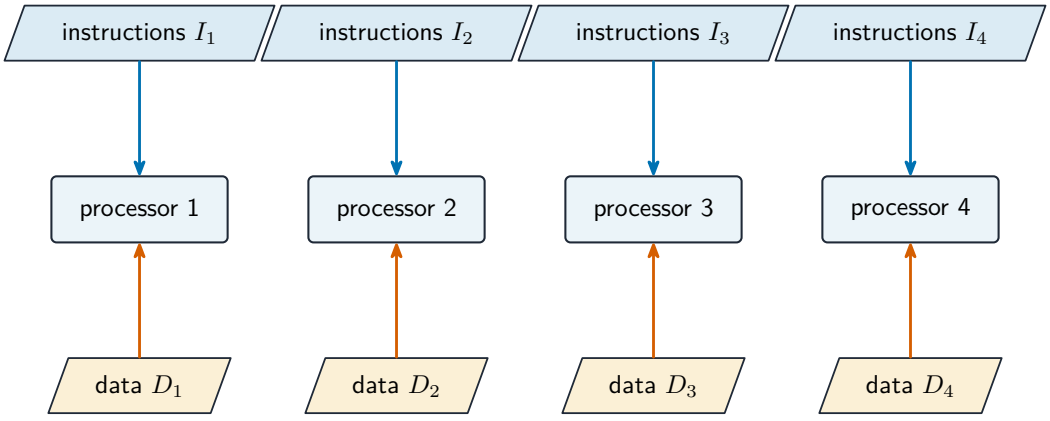
- **SISD** — one instruction, one data stream (a traditional single core).
- **SIMD** 单指令多数据 — one instruction works on **many data items at once** (GPUs, CPU vector extensions). Great for images, video, scientific arrays.
- **MISD** — several operations on the same data; rare, mostly theoretical.
- **MIMD** 多指令多数据 — many processors run **different instructions on different data** (multi-core CPUs, clusters). The most general.




*SIMD: many processors run the same instruction on different data*

A **graphics card** 显卡 (with its GPU) is a real example of SIMD hardware: it has thousands of small cores that run the same instruction on many pixels or numbers at once, which is why GPUs are so fast for images, video and machine learning.


*A graphics card: its GPU runs the same instruction on many data items at once (SIMD)*




*MIMD: each processor runs its own instructions on its own data*

## Massively parallel computers

A **massively parallel** 大规模并行 system uses **thousands of processors** on a fast network, each with its own memory (**distributed memory** 分布式内存), exchanging data by messages. It is MIMD, needs specially-written software (MPI, CUDA), and suits climate simulation, large **machine learning** 机器学习 training, and astrophysics. The largest **supercomputers** 超级计算机 are massively parallel.

The processors live in tall **server** 服务器 racks, often filling a whole room (a **data centre** 数据中心), wired together so they can work on one big problem at the same time.

*Rows of servers in a data centre, like those used for massively parallel computing*

# Virtual machines

A **virtual machine** 虚拟机 (VM) is a **software emulation of a whole computer** — the software inside sees a CPU, memory and disks that look real but are managed by host software.

- a **system VM** runs a complete OS. A **hypervisor** 虚拟机监控器 creates and manages VMs, each booting its own guest OS. Uses: run different OSes on one machine; server consolidation; **sandboxing** 沙箱 (risky software runs isolated); snapshots.
- a **process (language) VM** runs one program in portable **bytecode** 字节码 — the JVM (Java), the CLR (.NET), CPython. Benefits: portability ("write once, run anywhere"), runtime safety checks, and **just-in-time compilation** 即时编译 for near-native speed. The cost is an extra layer and needing the VM installed.

# Boolean algebra

**Boolean algebra** 布尔代数 simplifies **Boolean** 布尔 expressions, which can equally be described by **truth tables** 真值表. Symbols: `+` for OR, $\cdot$ for AND (often omitted), an overbar for NOT.

Key laws include commutative, associative and distributive (as in ordinary algebra), plus:

- identity $A + 0 = A$, $A \cdot 1 = A$; null $A + 1 = 1$, $A \cdot 0 = 0$.
- idempotent $A + A = A$; inverse $A + \overline{A} = 1$, $A \cdot \overline{A} = 0$.
- **De Morgan's laws** 德摩根定律: $(A + B)' = A' \cdot B'$; $(A \cdot B)' = A' + B'$ — negate the whole, swap AND/OR, negate each operand.
- **absorption** 吸收律: $A + AB = A$.

Simplifying reduces the number of terms, so the resulting logic circuit has fewer gates. Example: $Z = AB + A\overline{B} = A(B + \overline{B}) = A$.

# Karnaugh maps

A **Karnaugh map** 卡诺图 (K-map) simplifies a Boolean expression by **grouping adjacent 1s** from a truth table. Columns and rows use **Gray code** 格雷码 order (`00`, `01`, `11`, `10`) so adjacent cells differ in one variable.

Place a 1 in each cell where the output is 1. Find rectangular groups of 1s whose sides are powers of 2 (1, 2, 4, 8), wrapping around edges if it makes a bigger group. **The larger the group, the simpler the term**: a group of 2 drops one variable, a group of 4 drops two, and so on — variables that change within the group disappear. OR

the group terms together for the simplified expression. Cover every 1 using as few, as large, groups as possible.

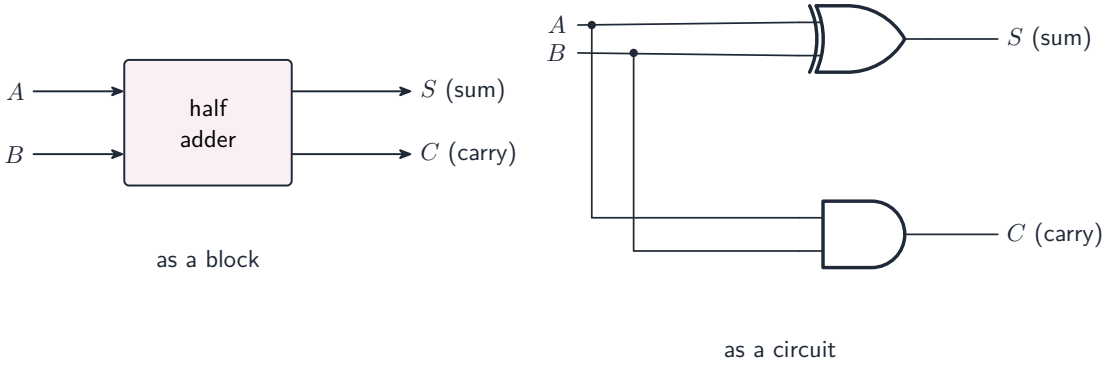
**Worked example.** A Karnaugh map for $A$ and $B$ has 1s in the cells $\overline{A}B$ and $AB$. Simplify. The two 1s are **adjacent** - they share the $B = 1$ column - so group them as a rectangle of 2. Inside that group $B$ stays 1 throughout while $A$ **changes** from 0 to 1, and any variable that changes within a group **disappears**. So the group leaves simply $X = B$. Compare that with the sum of products read straight off the table, $\overline{A}B + AB$: the same circuit, two gates fewer. Two rules do most of the work - make each group **as large as possible** (a group of 2 drops one variable, 4 drops two, 8 drops three), and remember the map **wraps around** its edges, so the leftmost and rightmost columns are adjacent. That wrap is the grouping most candidates miss.

# Half adder and full adder

A **half adder** 半加器 adds two single bits $A$ and $B$, giving a sum $S$ and a **carry** 进位 $C$:

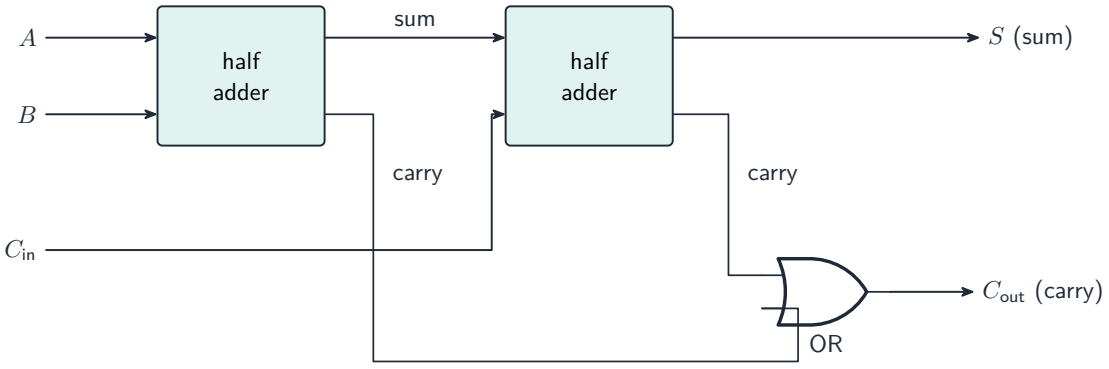
| A | B | S | C |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 |

So $S = A$ XOR $B$ and $C = A$ AND $B$. It ignores any carry-in — hence "half".




*A half adder, as a block and as a circuit of an XOR and an AND gate*

A **full adder** 全加器 adds three bits ($A$, $B$, carry-in), giving a sum and a carry-out: $S = A$ XOR $B$ XOR $C_{\text{in}}$. It can be built from two half adders plus an OR gate. Chaining full adders (each carry-out feeding the next carry-in) makes a multi-bit "ripple-carry" adder.




*A full adder is built from two half adders and an OR gate*

# Flip-flops

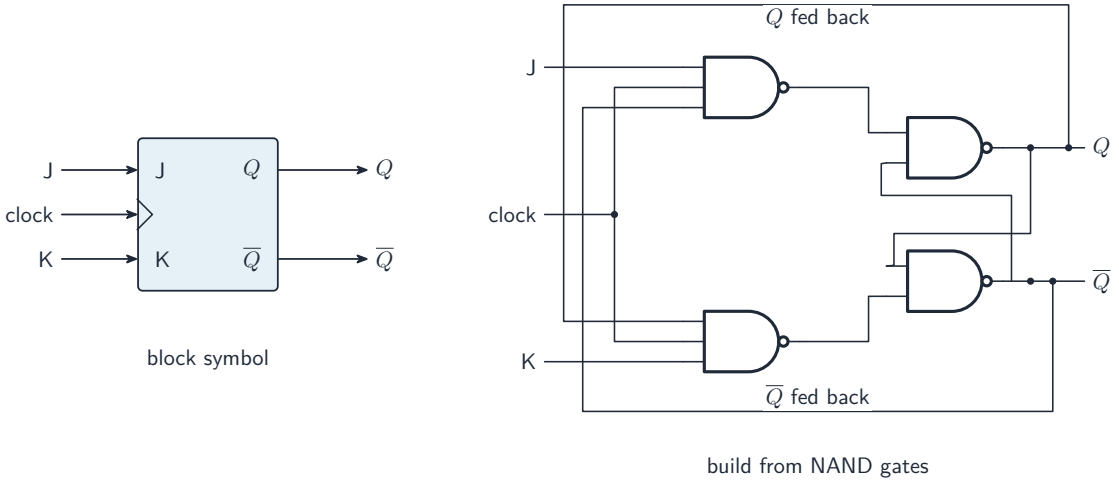
A **flip-flop** 触发器 is a **bistable** 双稳态 circuit — two stable states (0 and 1) — that **remembers** its state. It stores one bit and is the basic element of registers and SRAM.

## SR flip-flop

An **SR flip-flop** SR 触发器 has inputs **S** (set) and **R** (reset) and outputs Q and $\overline{Q}$. `S=1,R=0` sets Q to 1; `S=0,R=1` resets it to 0; `S=0,R=0` holds; `S=1,R=1` is **invalid**. Built from two cross-coupled NOR gates.

### JK flip-flop

A **JK flip-flop** JK 触发器 improves on it by using the previously-invalid `1,1` input as a **toggle** 翻转 (the output flips). This makes it ideal for building **counters** 计数器 (a chain of toggling flip-flops). It is usually **clocked** — inputs act only on a clock edge, keeping flip-flops synchronised.




*A JK flip-flop: its symbol and a build from NAND gates*

Flip-flops are the building blocks of registers (n bits = n flip-flops), counters, and **SRAM** 静态 RAM cells.

## Exam tips

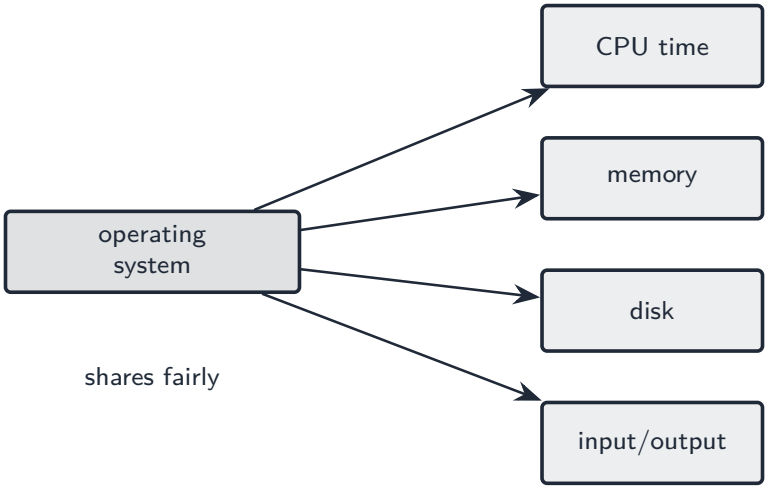
- Compare **RISC vs CISC** (simple, fast, uniform instructions vs complex ones) and why RISC suits pipelining.
- Simplify logic with **Boolean algebra / Karnaugh maps** — group the 1s in powers of two.
- Explain a **half adder vs full adder** (the full adder handles a carry-in) and what a flip-flop stores.
- Place a machine in **Flynn's taxonomy** (SISD, SIMD, MISD, MIMD).

# System Software

A-Level Computer Science

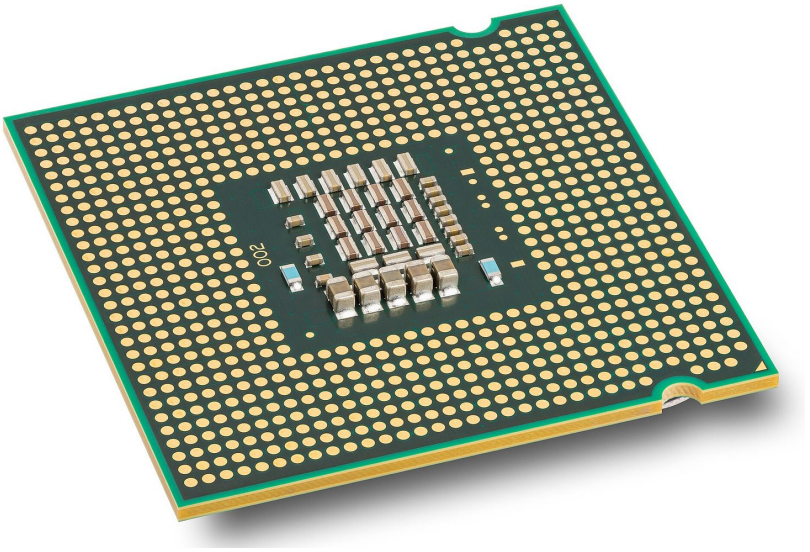
## How an OS maximises use of resources

A computer has many resources (CPU time, memory, disk, I/O) and many programs competing for them. The OS **shares them fairly and efficiently** so each is well used and the system stays responsive:




*The OS shares the CPU, memory, disk and I/O between programs*

- **multi-tasking** 多任务 — switch the CPU quickly between processes so several seem to run at once.
- **memory management** — give each process the memory it needs; use disk **paging** 分页 when **RAM** runs out.
- **spooling** 假脱机 and buffering — print jobs queue on disk so the CPU never waits for the printer.
- **caching** — keep recently-used disk data in **cache** 高速缓存 / RAM.



*The processor is a key resource the OS shares between competing tasks*


*The OS also manages memory (RAM), deciding what to keep in it and what to page out to disk*

## The user interface

The user interface hides the hardware behind friendly abstractions: the user sees windows, menus and folders, not addresses or sectors. One click on an icon makes the OS find the program on disk, allocate memory, load it and start it. A **CLI** (command line) is powerful and scriptable for experts; a **GUI** (graphical) is easier to learn. Most systems offer both.

# Process management

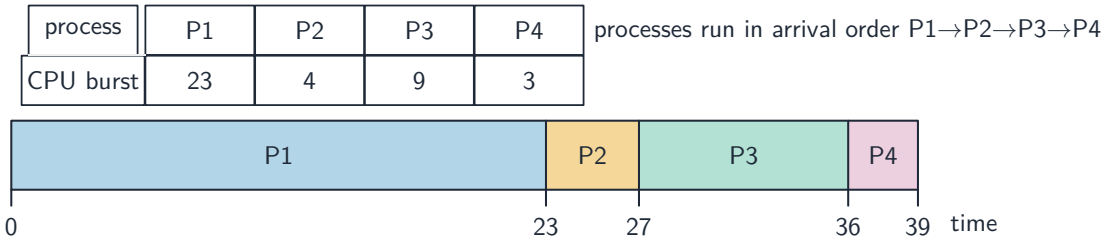
A **process** 进程 is a program in execution — its code, current state, memory and open files.

## Scheduling

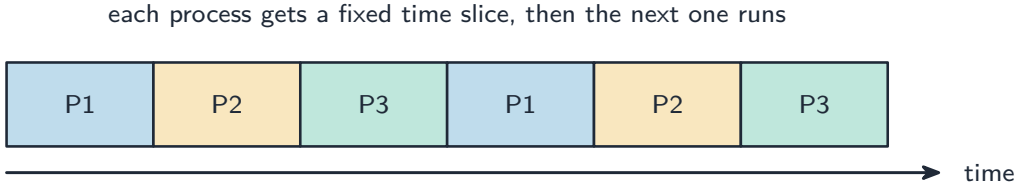
The **scheduler** 调度器 chooses which **ready** process runs next, and for how long:

- **round robin** 轮转 — each process gets a fixed **time slice** 时间片, then goes to the back of the queue.
- first-come-first-served; shortest job first; **shortest remaining time** (run the job with the least work left); priority; multilevel feedback queues.

The trade-off is responsiveness vs throughput vs fairness.



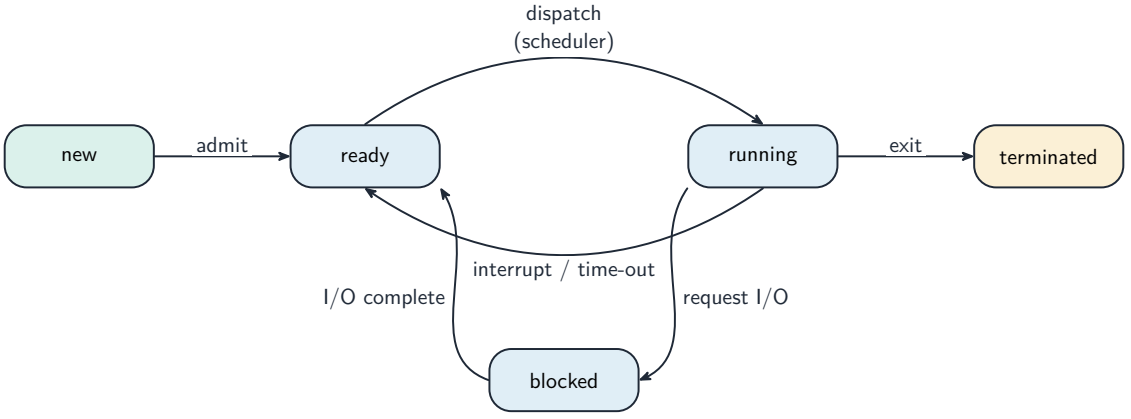

*First-come-first-served scheduling of four processes*




*Round-robin: each process gets a fixed time slice in turn, then the next runs (unlike first-come-first-served)*

## Process states

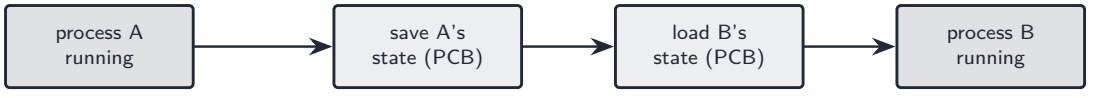
A process is **new**, **ready** (waiting for the CPU), **running**, **blocked** 阻塞 (waiting for I/O or a lock), or **terminated**. When its time slice ends it goes running → ready; when it requests I/O it goes running → blocked; when the I/O finishes it goes blocked → ready.




*A process moves between the new, ready, running, blocked and terminated states*

## Process control block and context switch

For each process the OS keeps a **process control block** 进程控制块 (PCB) — the saved program counter, registers, state and memory info.




*A context switch saves one process's state and loads another's*

- a **context switch** 上下文切换 suspends one process and starts another: it saves the state into one PCB and restores it from another. This small cost is paid on every switch.
- the **kernel** 内核 (the core of the OS) acts as an **interrupt handler** 中断处理程序. When a device or the timer raises an interrupt, **interrupt handling** 中断处理 saves the running process and runs the right routine — this is what drives low-level scheduling.

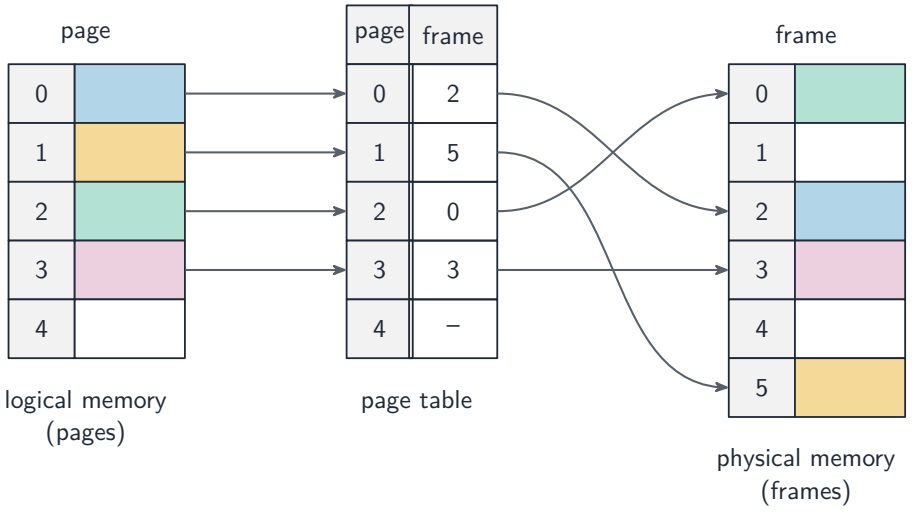
### Inter-process communication

Processes are isolated, so the OS provides **inter-process communication** 进程间通信: **pipes** 管道 (one program's output feeds another's input), **shared memory** 共享内存 (a region several processes can use), and message passing.

## Virtual memory, paging, segmentation

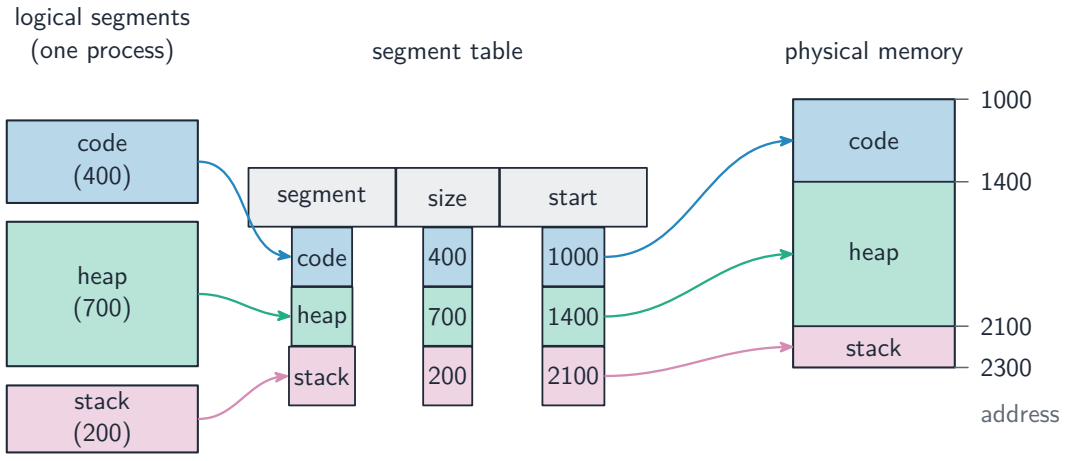
Each process gets its own **virtual address space** 虚拟地址空间 — a clean, contiguous range of addresses the OS maps to physical memory. This gives each process a simple space, **protects processes from each other**, and lets the total memory **exceed physical RAM**.

In **paging**, the virtual space is split into fixed-size **pages** 页 and physical memory into same-sized **frames** 页框. A page table maps each page to a frame. If an accessed page is not in RAM — a **page fault** 缺页 — the OS reads it from the **swap file** 交换文件 into a frame, evicting another page if RAM is full. Frequent faults cause **thrashing** 抖动 (**disk thrashing**), where the OS spends most of its time swapping pages instead of doing useful work.




*Paging maps each page of logical memory to a frame of physical memory*

In **segmentation** 分段, memory is split into variable-sized logical segments (code, stack, heap), each with its own permissions. Many systems use paging within segments.




*Segmentation maps variable-sized segments using a segment map table*

## How an interpreter runs a program

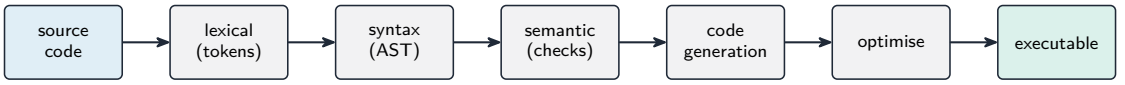
An **interpreter** 解释器 translates and runs the source **at the same time**. For each statement it reads the line, does lexical and syntax analysis, checks types, then **executes** the action, and moves on. Errors are reported immediately and it usually stops; no executable is produced. The translation is redone every run (slower), but it gives fast development feedback and is portable.

## Stages of compilation

A **compiler** 编译器 turns source into **machine code** 机器码 in phases:

1. **lexical analysis** 词法分析 — the lexer groups characters into **tokens** 词法单元 (keywords, identifiers, operators, literals), discarding whitespace and comments.
2. **syntax analysis (parsing)** 语法分析 — check the tokens fit the grammar and build an **abstract syntax tree** 抽象语法树. A missing bracket gives a **syntax error** 语法错误.
3. **semantic analysis** 语义分析 — check the program makes sense (variables declared, types match).
4. **code generation** 代码生成 — walk the tree and emit target code, choosing registers and layouts.
5. **code optimisation** 代码优化 — remove redundant work, fold constants, reorder for the pipeline.

The output is an executable.




*The phases of compilation, from source code to an optimised executable*

# Grammar: BNF and syntax diagrams

A **grammar** 文法 says which token sequences are valid programs.

**Backus-Naur Form** 巴科斯-诺尔范式 (BNF) is textual. A **production rule** 产生式 has the form:

```
<symbol> ::= alternative1 | alternative2 | ...
```

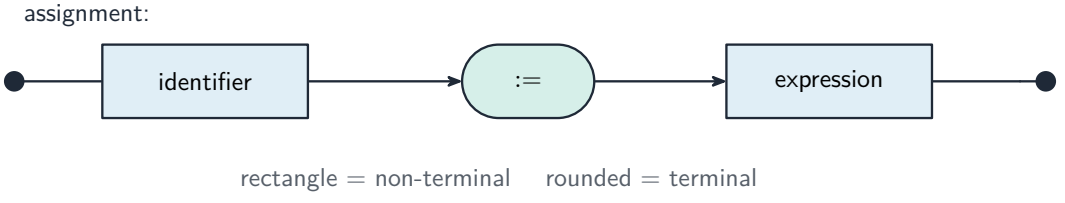
Each alternative is a sequence of **terminal** 终结符 symbols (literal text) and **non-terminal** 非终结符 symbols (other rule names):

```
<digit>      ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<identifier> ::= <letter> | <identifier> <letter> | <identifier> <digit>
```

The recursive third rule expresses "a letter followed by any number of letters or digits". An `IF` statement:

```
<if-statement> ::= IF <condition> THEN <statement> ENDIF
                 | IF <condition> THEN <statement> ELSE <statement> ENDIF
```

A **syntax diagram** 语法图 (railroad diagram) shows the same thing graphically: boxes for non-terminals, rounded boxes for terminals, arrows for valid paths, loops for repetition. The two notations are equivalent. The parser uses the grammar to decide whether a program is valid.




*A syntax (railroad) diagram for an assignment statement*

# Reverse Polish Notation (RPN)

In **infix** 中缀 notation the operator sits between its operands (`3 + 4 * 2`), needing brackets and precedence rules. In **Reverse Polish Notation** 逆波兰表示法 (RPN, **postfix** 后缀) the operator follows its operands (`3 4 2 * +`), needing no brackets.

## Converting infix to RPN

Use an operator **stack** 栈. Scan left to right: output an operand; for an operator, first pop any stacked operators of **higher or equal precedence** 优先级 to the output, then push it; push `(`; on `)` pop to output until the matching `(`. At the end, pop all operators. Example: `(3 + 4) * 2` → `3 4 + 2 *`.

## Evaluating RPN

Use a stack of operands. Scan left to right: push each operand; on an operator, pop the top two, apply it, and push the result. Evaluating `3 4 2 * +`:

| Token | Stack |
|---|---|
| `3` | 3 |
| `4` | 3, 4 |
| `2` | 3, 4, 2 |
| `*` | 3, 8 |
| `+` | 11 |

Result: 11. RPN needs no brackets at evaluation time and suits a stack machine — which is how the JVM and many **bytecode** 字节码 interpreters work.

**Worked example.** Convert $(A+B)\times(C-D)$ to RPN, then evaluate $(3+4)\times(5-2)$. Scan left to right using an operator **stack**. Push `(`; output `A`; push `+`; output `B`; on `)` pop back to the matching `(`, giving `A B +` so far. Push `×`, and the second bracket behaves the same way, giving `C D -`. At the end pop the `×`. Result: **`A B + C D - ×`**. To evaluate the numbers, use a stack of **operands**: push 3, push 4; `+` pops both and pushes 7; push 5, push 2; `-` pops both and pushes 3; `×` pops 7 and 3 and pushes **21**. Two things make these reliable: the operands keep their **original order** through the conversion (only the operators move), and every operator acts on the **two values immediately below it** on the stack.

# Exam tips

- List the **stages of compilation** (lexical, syntax and semantic analysis, code generation, optimisation) and what each does.
- Explain **virtual memory and paging** (disk used as extra RAM) and its cost (disk thrashing).
- Evaluate **Reverse Polish Notation** with a stack — no brackets are needed.
- Use **BNF or a syntax diagram** to test whether a string is valid.

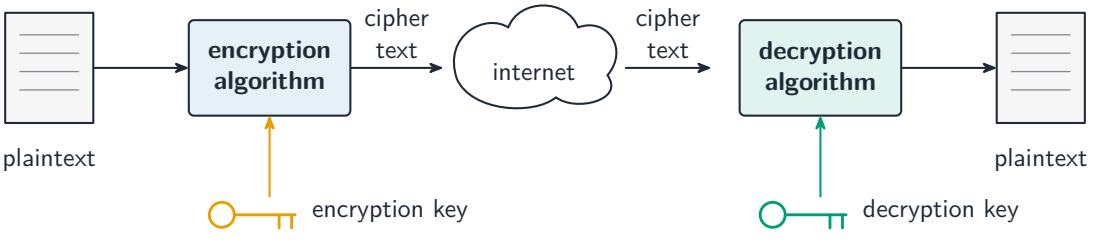
# Security

A-Level Computer Science

## How encryption works

**Encryption** 加密 turns readable **plaintext** 明文 (plain text) into unreadable **ciphertext** 密文 (cipher text) using a maths operation that depends on a **key**. Only someone with the right key can reverse it — **decryption** 解密 — to get the plaintext back. An attacker who intercepts the ciphertext without the key sees only meaningless data, because trying every possible key would take far too long. A newer approach, **quantum cryptography** 量子密码学, uses quantum physics to share a key in a way that reveals any eavesdropper.

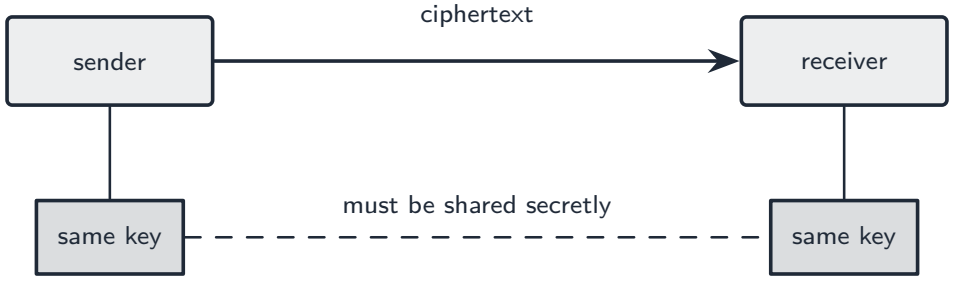
*The Enigma machine encrypted messages in the Second World War — an early, mechanical cipher device*




*Encryption scrambles plaintext with a key; decryption reverses it*

## Symmetric encryption

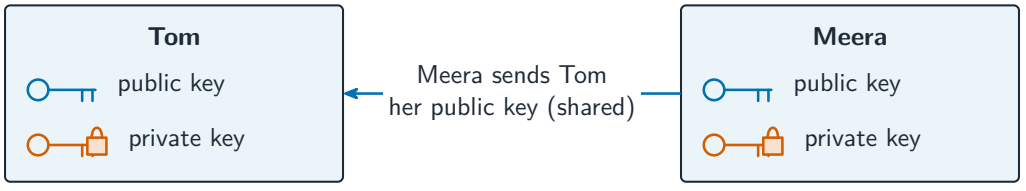
**Symmetric encryption** 对称加密 (symmetric key cryptography) uses the **same key** for both encryption and decryption, so sender and receiver must both hold the secret key. It is **fast** and good for **bulk data** (a whole disk, a video stream). Its problem is **key distribution** 密钥分发: how do you share the key safely in the first place? Asymmetric encryption solves this.




*Symmetric encryption uses the same secret key at both ends*

## Asymmetric encryption (public-key)

**Asymmetric encryption** 非对称加密 (asymmetric key cryptography) gives each user a **pair** of related keys: a **public key** 公钥 they publish, and a **private key** 私钥 they keep secret. Data encrypted with the public key can be decrypted **only** with the matching private key, and vice versa.




encrypt with the recipient's **public key** — only their matching **private key** can decrypt

*Each user has a public key to share and a private key to keep secret*

To send a secret message to Alice: get her published public key, encrypt with it, and send. Only Alice — holding the matching private key — can decrypt. No prior key exchange is needed. The trade-off is that it is **much slower** than symmetric, so it is not used for large data.

A private key must stay secret, so it is sometimes kept on a small **hardware security key** 硬件安全密钥. You plug it in or tap it to prove who you are, and the secret key never leaves the device.

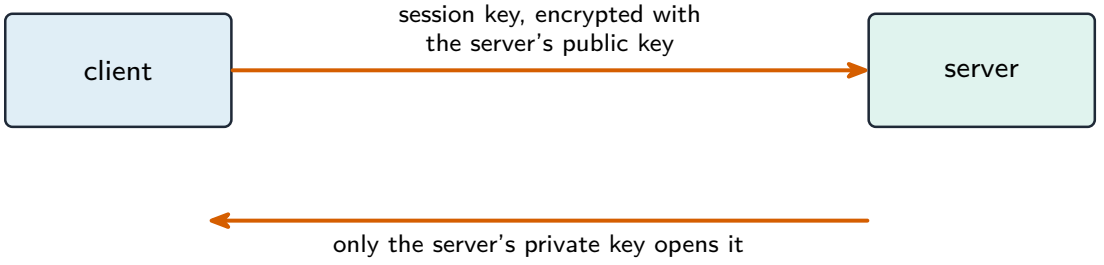
*A hardware security key stores a secret key to prove who you are*

## Hybrid approach (used by almost every real system)

Use asymmetric encryption to exchange a fresh **session key** 会话密钥, then use that symmetric key for the data:

1. the client makes a random session key.
2. it encrypts the session key with the server's public key.
3. the server decrypts it with its private key.
4. both ends now share the session key and use fast symmetric encryption for the rest.

This is how HTTPS and SSH work.

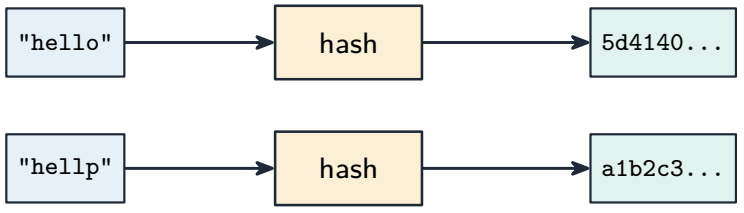



both ends now share the session key → fast **symmetric** encryption for all the data

*The hybrid approach: asymmetric crypto shares a session key once, then fast symmetric encryption protects the data*

### Hashing (related, not encryption)

A **cryptographic hash** 密码散列 function takes any input and gives a fixed-size **digest** 摘要 such that the same input always gives the same digest, it is infeasible to find two inputs with the same digest, and a tiny change in input changes the digest completely. Hashing is **one-way** — you cannot get the input back. It is used for storing password checks, **integrity** 完整性 checks, and digital signatures.




one letter changed → a completely different digest; one-way (cannot reverse)

*A cryptographic hash gives a fixed digest; a tiny input change changes it completely, and it cannot be reversed*

## SSL / TLS

**TLS** 传输层安全 (Transport Layer Security, the successor to the **Secure Socket Layer**, SSL) is a **protocol** that gives encryption and authentication for data sent over a network. It **encrypts** the data in transit, **authenticates** the server with a certificate, and provides **integrity** (detecting tampering).

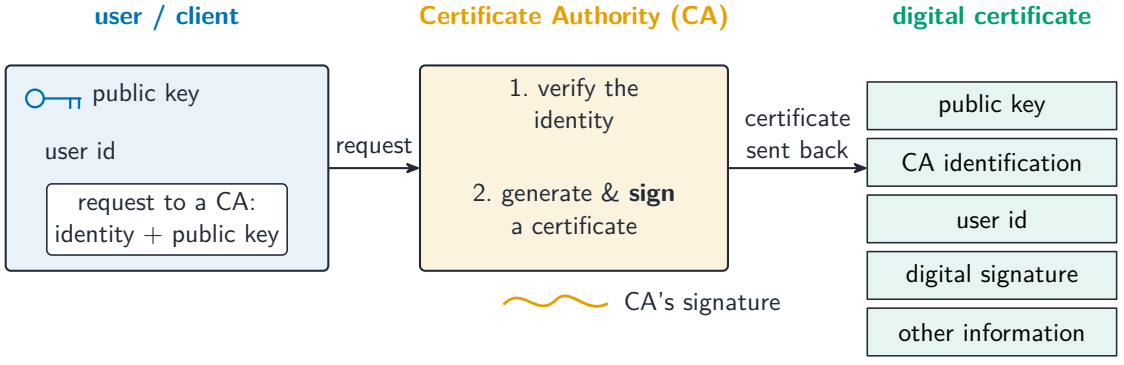
Outline of a TLS handshake:

1. the client connects and proposes cipher options.
2. the server picks one and sends its **digital certificate** (with its public key) — issuing and validating these certificates is **digital certification**.
3. the client checks the certificate.
4. the two ends exchange a fresh **session key** using asymmetric crypto.
5. all later traffic uses fast symmetric encryption with the session key.

The result is an encrypted, authenticated, integrity-checked tunnel for higher-level protocols (HTTP, SMTP). It is appropriate wherever **sensitive information** is sent: HTTPS web browsing, online banking and payments, secure email, and VPNs.

# Digital certificates

A **digital certificate** 数字证书 binds an identity (a domain, an organisation) to a public key, and is signed by a trusted **Certificate Authority** 证书颁发机构 (CA). It contains the subject (who it identifies), the subject's public key, the issuer (the CA), a validity period, and the CA's signature over all of it.




*A Certificate Authority issues a digital certificate binding an identity to a public key*

To verify one, the client (which holds a list of trusted root CAs):

1. checks the expiry dates.
2. checks the subject name matches the URL.
3. checks it is **signed by a trusted CA**, using the CA's public key to verify the signature.
4. follows the certificate chain up to a trusted root.

If anything fails, the browser shows the "Your connection is not private" warning. When it verifies cleanly, the client knows the identity was vetted by a trusted CA, the public key really belongs to that identity, and the certificate is current.

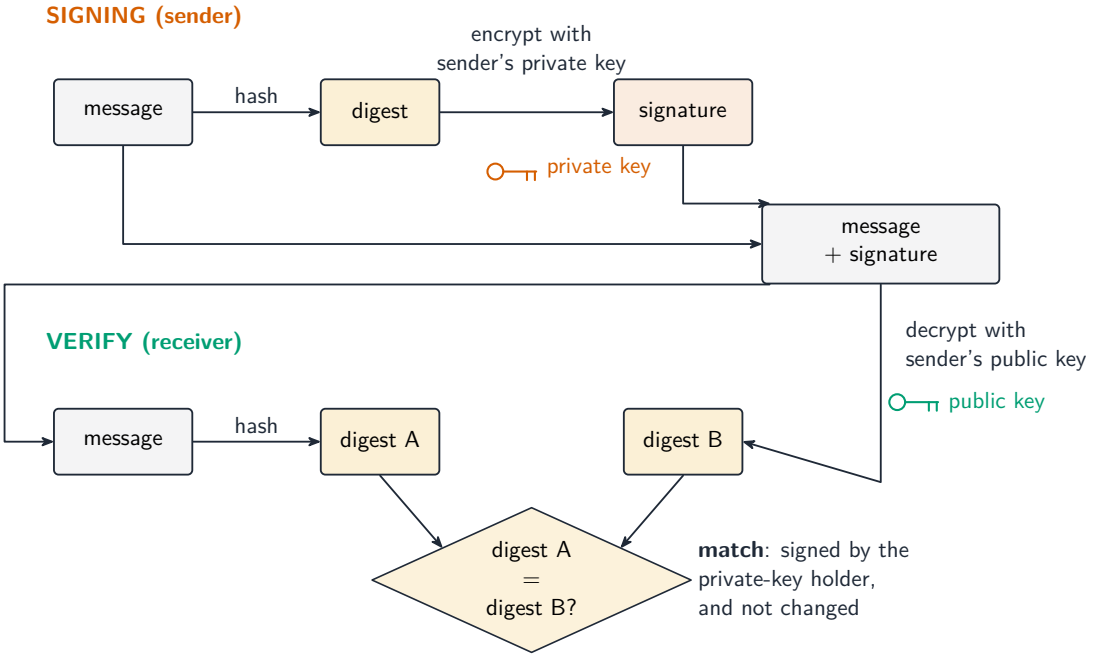
# Digital signatures

A **digital signature** 数字签名 proves who signed a message **and** that it was not changed. To sign:

1. compute a cryptographic hash of the message.
2. **encrypt the hash with the sender's private key** — that is the signature.
3. send the message and the signature.

To verify: compute the hash of the received message; **decrypt the signature with the sender's public key** to get the sender's hash; compare. If they match, the mes-

sage was signed by the holder of the private key (**authentication** 身份验证) and was not changed (integrity). A signature does **not** hide the message — for confidentiality as well, encrypt **and** sign.




*Signing hashes the message and encrypts the digest with the private key; the receiver checks it with the public key*

## Putting it together

A secure request to `https://www.bank.com`: the server sends its certificate; the client verifies it against trusted CAs; the client uses the server's public key to exchange a session key; then data flows encrypted with that key. Encryption stops eavesdroppers, the certificate proves the server's identity, and integrity checks stop a **man-in-the-middle** 中间人攻击 altering the data.

**Worked example.** Alice sends Bob a contract. She wants Bob to be certain it came from her and was not altered, **and** she wants nobody else to be able to read it. Which keys does she use, and in which direction? These are two different jobs needing two different key pairs. For the **signature** (authentication and integrity): Alice hashes the contract and encrypts that hash with **her own private key**; Bob decrypts it with **Alice's public key** and compares it against his own hash of the message. Only Alice holds her private key, so only she could have produced it. For **confidentiality**: Alice encrypts the contract itself with **Bob's public key**, so only **Bob's private key** can open it. One rule keeps all four straight: you **sign with your own private key** and **encrypt with the recipient's public key**. A signature on its own does **not** hide

the message.

## Exam tips

- Distinguish **symmetric** encryption (one shared key, fast) from **asymmetric** (a public/private key pair).
- Explain the **TLS handshake** and why a **digital certificate** from a CA proves identity.
- Explain a **digital signature**: hash the message, then encrypt the hash with the private key — it proves integrity and origin.

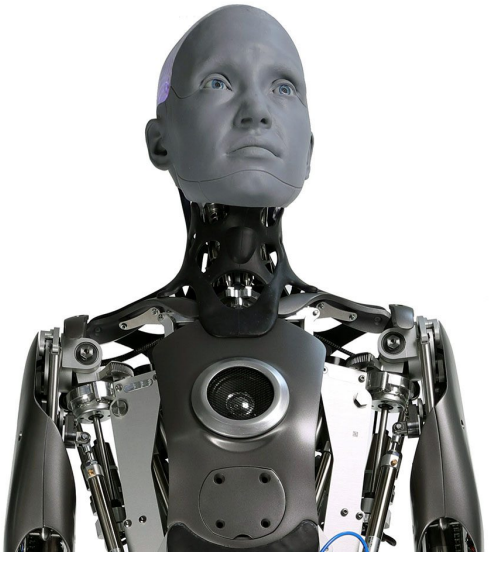
# Artificial Intelligence (AI)

A-Level Computer Science

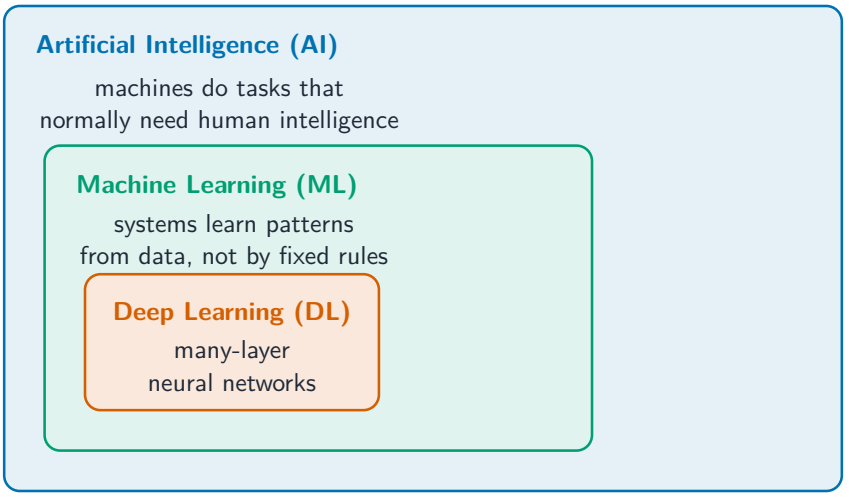
## What AI is

**Artificial intelligence** 人工智能 (AI) builds systems that do tasks normally needing human intelligence — recognising speech and images, translating, playing games, driving, generating text. Most modern AI uses **machine learning** 机器学习 — algorithms that learn patterns from data instead of being programmed step by step. Within it, **deep learning** 深度学习, using **neural networks** 神经网络 with many layers, has been dominant since the 2010s.

A **humanoid robot** 人形机器人 puts many of these abilities into one body: it uses AI to see faces, understand speech and move its face and arms in a lifelike way.



*A humanoid robot uses AI to see, listen and respond like a person*

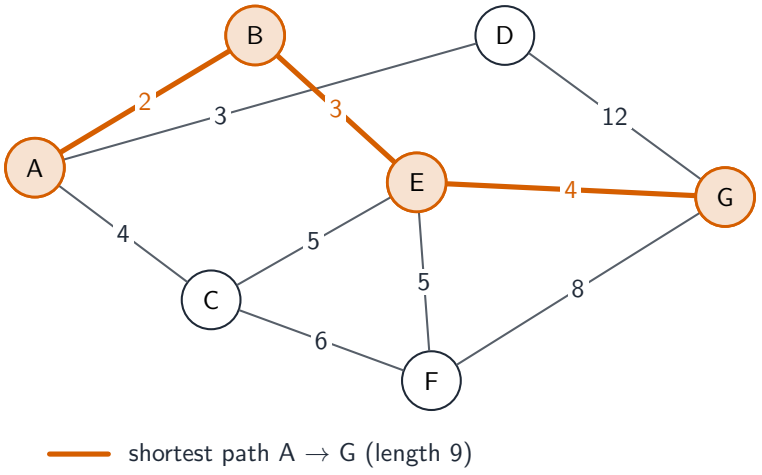



*Deep learning is part of machine learning, which is part of AI*

# Graphs in AI

Many AI problems sit on a **graph** 图 — **nodes** 节点 (states, places) joined by **edges** 边 (moves, relationships).

- **pathfinding**: roads form a graph; the shortest route is a graph search (**Dijkstra's algorithm**, the **A* algorithm**).
- **game playing**: each board position is a node, each move an edge; **minimax** 极小化极大 with alpha-beta pruning searches the game tree.
- **state-space search**: a planning problem is moving between states by applying operators to reach a goal.
- **knowledge representation**: a **semantic network** 语义网络 has concepts as nodes and relationships as edges ("dog IS-A animal"); a **knowledge graph** 知识图谱 stores facts about the world for search engines and assistants.



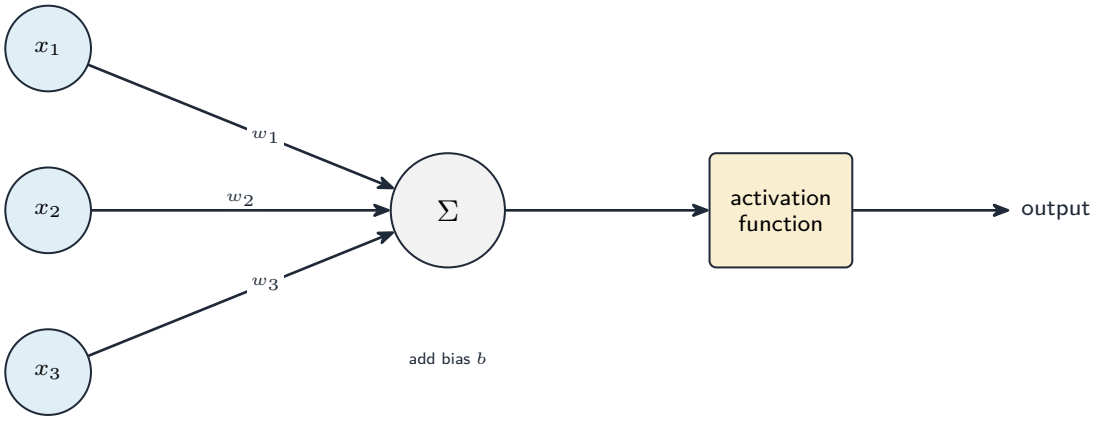

*AI problems often sit on a graph; here the shortest path is highlighted*

Standard tools for navigating graphs include **breadth-first search** 广度优先搜索 and **depth-first search** 深度优先搜索.

# Artificial neural networks (ANNs)

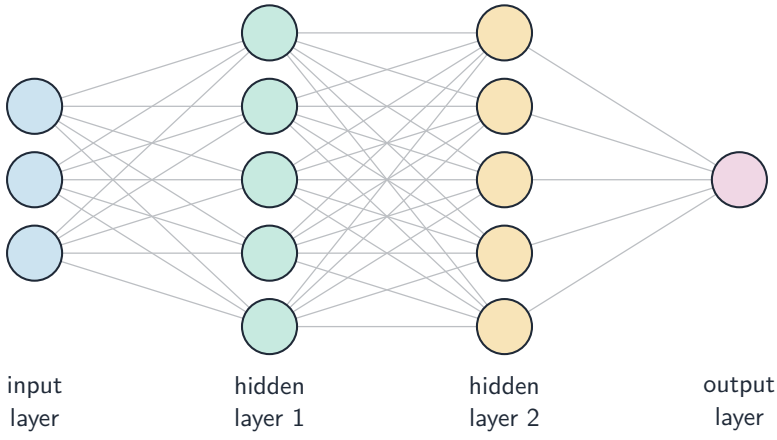
An ANN is inspired by the brain's neurons. An **artificial neuron** 人工神经元:

- takes several **input values**, multiplies each by a **weight** 权重, and adds them up with a **bias term** 偏置项.
- applies an **activation function** 激活函数 (a non-linear function such as ReLU) to the sum.
- outputs the result, which feeds neurons further on.




*A single neuron: each input times its weight, summed with a bias, then an activation function*

Neurons sit in layers: an **input layer**, one or more **hidden layers** 隐藏层 (where useful internal patterns are learned), and an **output layer**. With many hidden layers it is a **deep neural network** 深度神经网络, and training it is deep learning.




*A neural network with an input layer, two hidden layers and an output layer*

ANNs let models learn complex patterns straight from raw data (pixels, audio, text) without hand-designed features — driving breakthroughs in **image recognition** 图像识别, **speech recognition** 语音识别, **machine translation** 机器翻译, and game playing. They do well with large amounts of data, noisy or very complex input, and patterns too hard to capture with explicit rules.

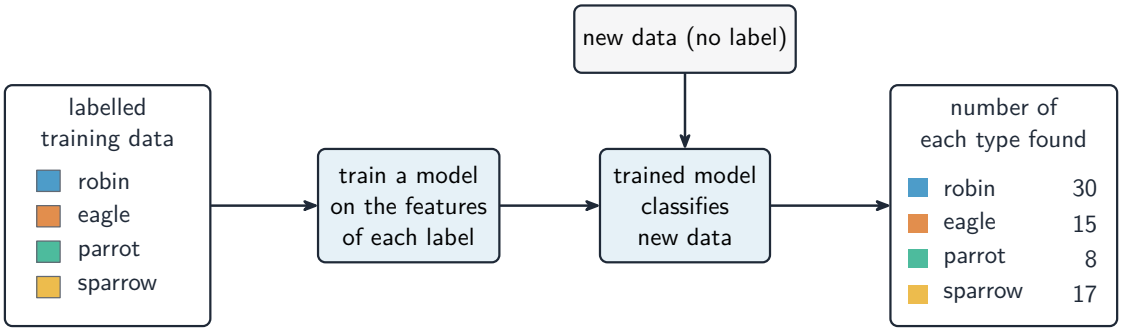
# Machine learning, deep learning, reinforcement learning

## Machine learning

The umbrella term — any algorithm that learns from data. Three paradigms:

- **supervised learning** 监督学习 — the data has **labels** 标签 (images tagged "cat"/"dog"); the algorithm learns input → label. Used for **classification** 分类 (a category) and regression.
- **unsupervised learning** 无监督学习 — no labels; the algorithm finds structure, e.g. a **cluster** 聚类 of similar customers.
- reinforcement learning (below).

Use ML when explicit rules would be impractical (spam filters, recommendations, fraud detection).



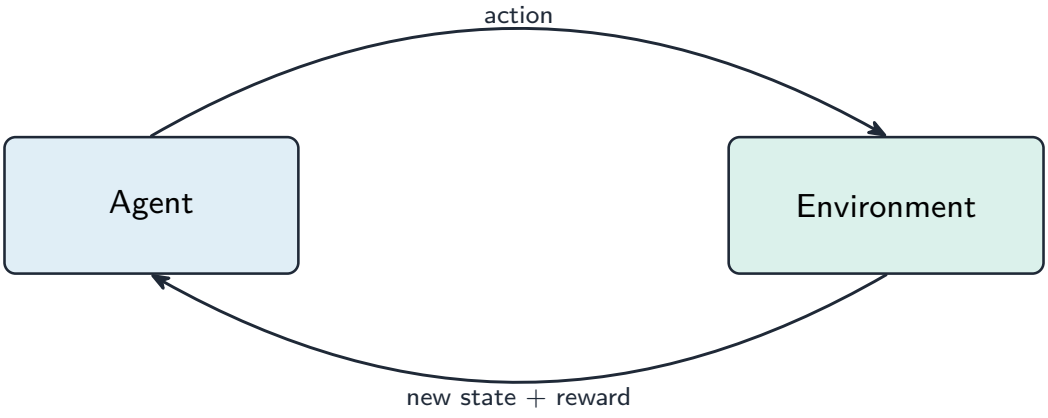

*Supervised learning: a model is trained on labelled data, then recognises new data*

## Deep learning

A subset of ML using deep neural networks. Lower layers learn simple patterns (edges, phonemes), higher layers combine them into abstract concepts. It needs **lots of data** and **lots of compute** (GPUs); for small datasets, simpler ML methods often do better.

## Reinforcement learning

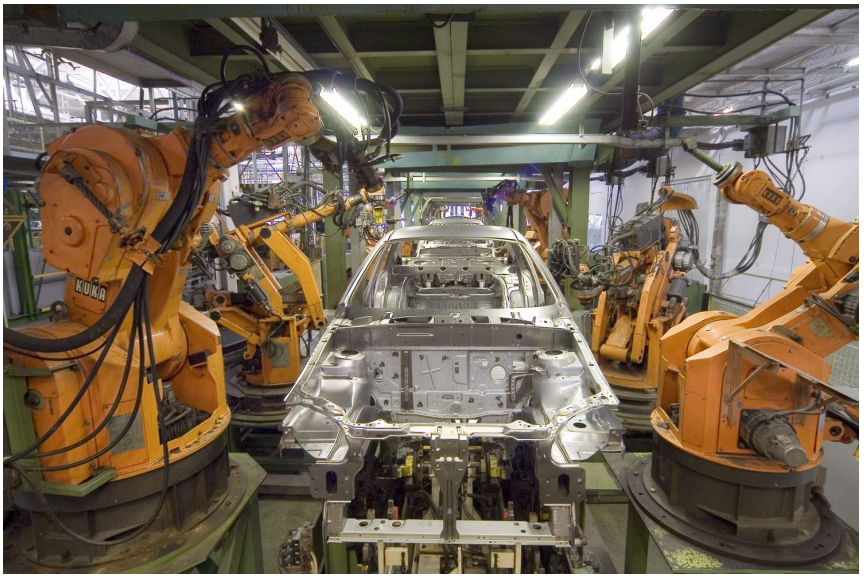
In **reinforcement learning** 强化学习, an **agent** 智能体 acts in an environment; each action changes the state and returns a **reward** 奖励. The agent learns a **policy** 策略 (a strategy) that maximises the total reward over time, by **trial and error** with no labels up front. Used for sequential-decision problems — games, robot control, autonomous driving.




*Reinforcement learning: the agent acts, the environment returns a new state and a reward, and the agent learns from it*

A **self-driving car** 自动驾驶汽车 is a real example. **Lidar** 激光雷达 and camera sensors (the spinning unit on the roof) build a live picture of the road, and a learned policy decides how to steer, speed up and brake safely.

*A self-driving car uses cameras and lidar sensors to see the road around it*



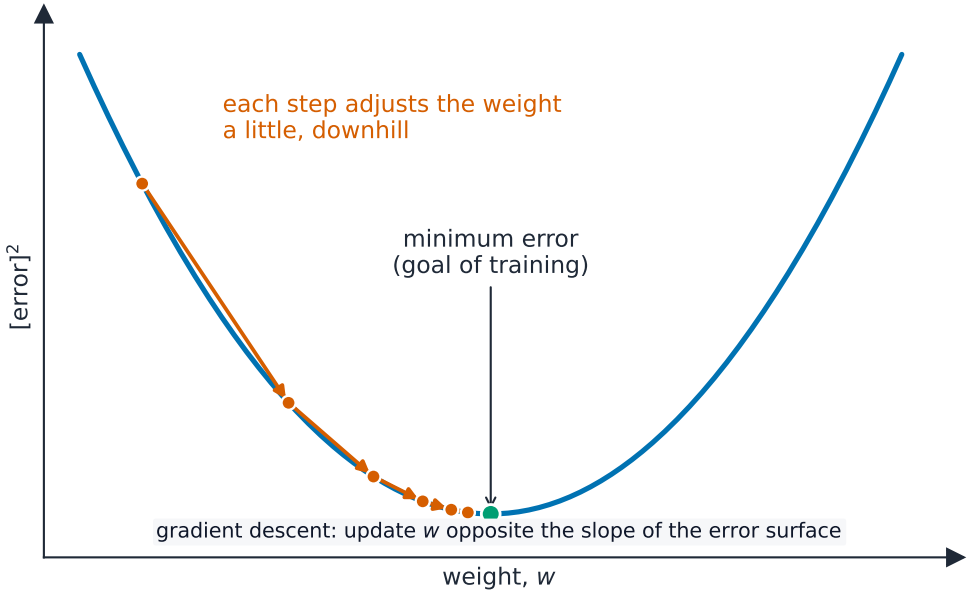
*Industrial robot arms on a production line: reinforcement learning can teach a robot to control its movements*

# Training an ANN: backpropagation

Training adjusts the **weights** so outputs match the targets. The standard method is **backpropagation** 反向传播 (back propagation of errors) with **gradient descent** 梯度下降. For each training example:

1. **forward pass** — feed the input through to the output.
2. **compute the error** with a **loss function** 损失函数 (a single number for how wrong the output is).
3. **backward pass** — propagate the error **backwards**, finding each weight's **gradient** (how much it contributed to the error) using the chain rule.
4. **update the weights** by a small step (set by the **learning rate** 学习率) that reduces the error.

Repeat over many examples and many passes (**epochs** 训练轮次) until the error stops shrinking. The name "back" comes from step 3: the error flows from the output **back** towards the input, so every weight's gradient is found in one sweep. After training, a new input needs only one forward pass to get a prediction.




*Training adjusts the weights to reach the minimum error*

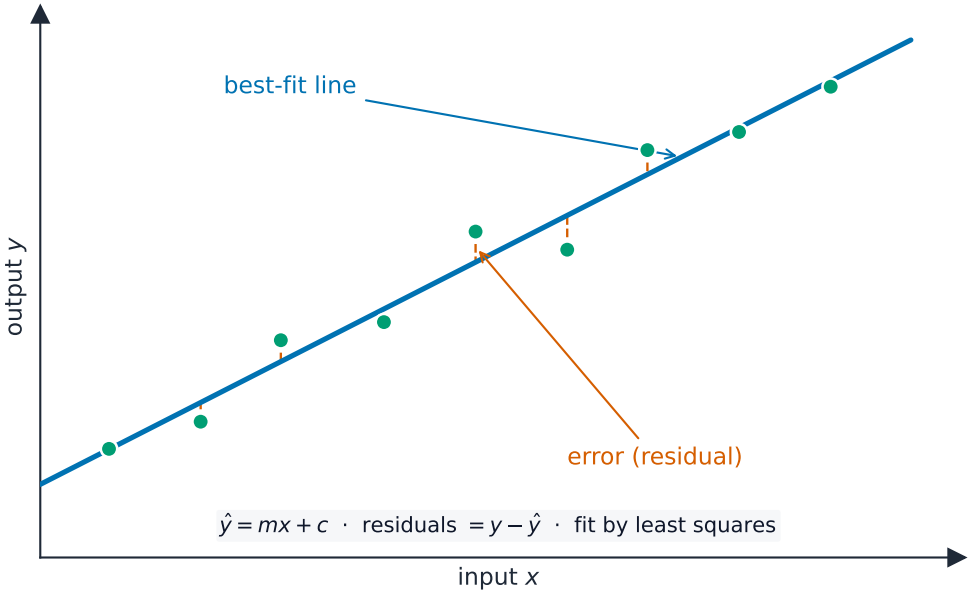
# Regression

Some tasks predict a **number** (a house price, tomorrow's temperature) — **regression** 回归, as opposed to classification (a category).

**Linear regression** 线性回归 fits a straight line (or hyperplane):

$$y = m_1x_1 + m_2x_2 + \ldots + m_nx_n + c.$$

Choose the coefficients to **minimise the sum of squared errors** against the training data. Use it when the relationship looks roughly linear and you want an interpretable model. For curved data, use polynomial, decision-tree, or neural-network **regression methods** — same idea: define a model, define a loss, and adjust the parameters to minimise it. Regression and classification are both supervised; the choice depends on whether the answer is a number or a category.




*Linear regression fits the line that makes the total squared error (the dashed gaps) as small as possible*

# How AI is used in a real scenario

Many exam scenarios use the same pattern — a **deep-learning model trained on labelled data**, often several combined into a pipeline:

- **customer identification** at an automated shop: the system is trained on labelled face images; a camera captures a face; image recognition extracts a representation; it is matched against registered customers; the closest match identifies the person.
- **reading text from images**: image recognition finds text regions; **optical character recognition** 光学字符识别 extracts the characters; machine translation converts them; **text-to-speech** 文本转语音 reads them aloud.
- **checkout item-detection**: object-detection AI, trained on labelled product images, sees which items go into a basket and charges the account.

By the time a user interacts with the system, the model is fast — it only does forward-pass inference; the intelligence is in the patterns learned during training.

**Worked example.** For each task, say whether it needs **regression** or **classification**, and what the output layer of an ANN would look like: (a) predict tomorrow's temperature; (b) decide whether an email is spam. Ask what **kind of thing** is being predicted. (a) A temperature is a **number** on a continuous scale, so this is **regression**, and the output layer is a **single** neuron holding that value. (b) Spam or not-spam is a **category**, so this is **classification**, and the output gives a probability per class. Both are **supervised** learning: each needs labelled examples to train on, and training adjusts the **weights** by **backpropagation** to reduce the error. The deciding question is simply number-or-category - not how difficult the task feels.

# Exam tips

- Distinguish **machine learning, deep learning and reinforcement learning** with an example of each.
- Describe an **ANN** (input, hidden and output layers; weighted connections) and how **backpropagation** adjusts weights to cut error.
- Distinguish **supervised vs unsupervised** learning; regression predicts a continuous value.

# Computational thinking and Problem-solving

A-Level Computer Science

## Searching algorithms

A **search** finds a **target value** in a collection (often an **array** 数组) and returns its position, or “not found”.

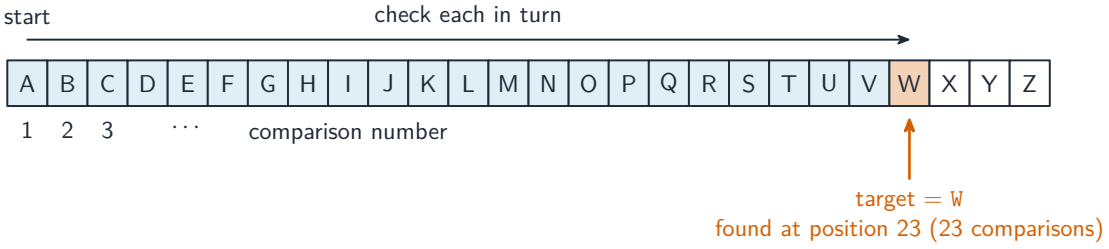

*Searching a sorted list, like a phone book, is far faster than checking every entry one by one*

### Linear search

A **linear search** 线性查找 walks from start to end, comparing each element with the target:

```
FOR i ← 1 TO n
    IF A[i] = target THEN RETURN i
NEXT i
RETURN -1   // not found
```

No preparation is needed, so it works on any list. Worst case O($n$) (target at the end or absent); best case 1 comparison. Use it on **unsorted** data or small lists. (The returned `-1` is a **sentinel** value — an impossible position that means ”not found”; the caller tests `IF result = -1`.)



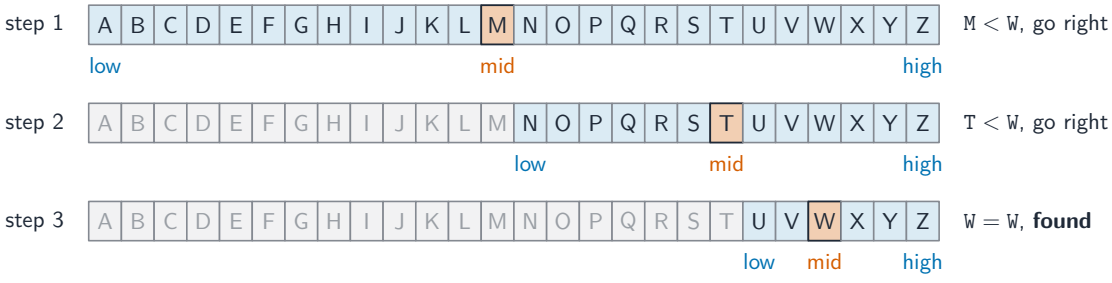

*Linear search checks every letter in turn — 23 comparisons to find W*

## Binary search

A **binary search** 二分查找 needs the data **sorted**. Look at the middle element; if it is the target, done; if the target is smaller, search the left half, else the right half — halving the range each time:

```
low ← 1
high ← n
WHILE low <= high DO
    mid ← (low + high) DIV 2
    IF A[mid] = target THEN RETURN mid
    IF A[mid] < target THEN
        low ← mid + 1
    ELSE
        high ← mid - 1
    ENDIF
ENDWHILE
RETURN -1
```

Worst case $O(\log_2 n)$ — for a million items, about 20 comparisons. Much faster than linear search on large sorted arrays, but you must sort first (a one-off $O(n \log n)$ cost), worth it if you search many times.




*Binary search halves the range each step (low / mid / high) — just 3 comparisons to find W*

# Sorting algorithms

## Bubble sort

A **bubble sort** 冒泡排序 repeatedly walks the array, swapping adjacent pairs that are out of order, so the largest "bubbles" to the end each pass:

```
FOR pass ← 1 TO n - 1
    swapped ← FALSE
    FOR i ← 1 TO n - pass
        IF A[i] > A[i + 1] THEN
            temp ← A[i]
            A[i] ← A[i + 1]
            A[i + 1] ← temp
            swapped ← TRUE
        ENDIF
    NEXT i
    IF swapped = FALSE THEN EXIT FOR   // already sorted
NEXT pass
```

Best case $O(n)$ (already sorted, with the early exit); average/worst $O(n^2)$. Simple but slow for large $n$.

## Insertion sort

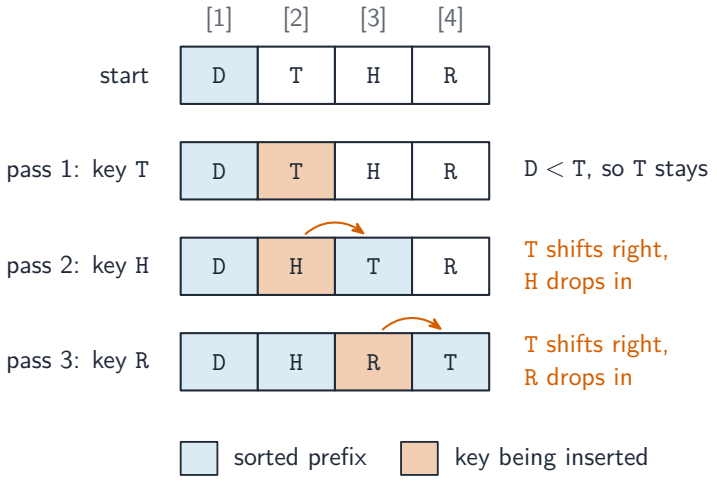
An **insertion sort** 插入排序 builds a sorted prefix from the left, inserting each new element into place by shifting larger ones right:

```
FOR i ← 2 TO n
    key ← A[i]
    j ← i - 1
    WHILE j >= 1 AND A[j] > key DO
        A[j + 1] ← A[j]
        j ← j - 1
    ENDWHILE
    A[j + 1] ← key
NEXT i
```

Best case $O(n)$ (already sorted); worst $O(n^2)$. Good for **small** or **nearly-sorted** arrays. It sorts **in place** 原地 and is **stable** 稳定 (keeps the order of equal elements).

## Tracing a sort

A common task is to show the array after each outer pass. For `[D, T, H, R]` with insertion sort: pass 1 (key T) no change; pass 2 (key H) → `[D, H, T, R]`; pass 3 (key R) → `[D, H, R, T]`.

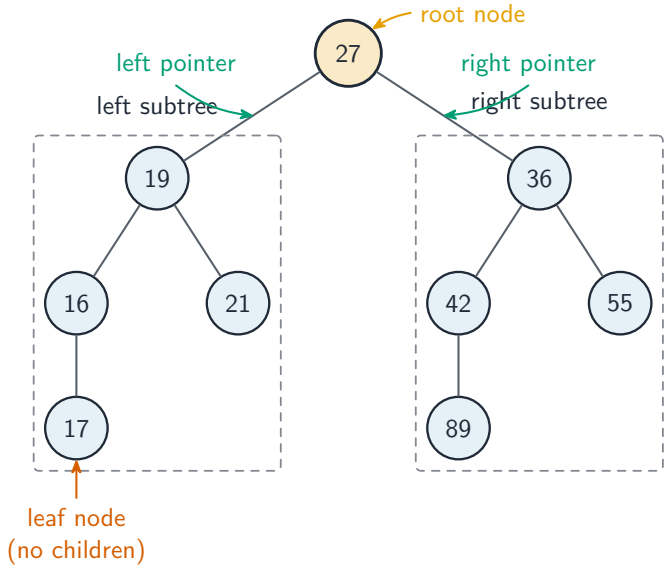



*An insertion sort of `[D, T, H, R]`, shifting each key into its place pass by pass*

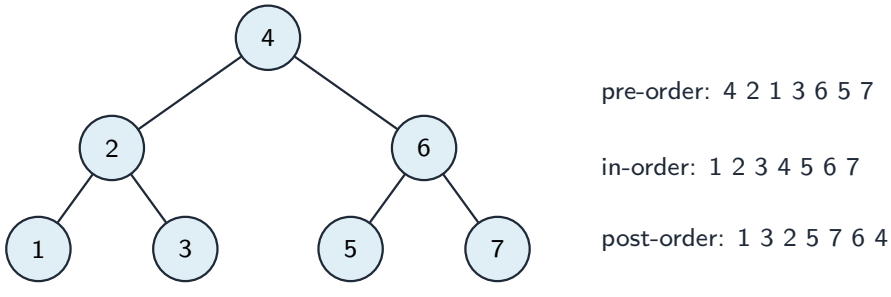
# ADTs in algorithms

The **Abstract Data Types** (ADTs) from Topic 10 appear inside many algorithms: a **stack** 栈 drives depth-first traversal and undo; a **queue** 队列 drives breadth-first traversal and print ordering; a **linked list** 链表 lets data grow and shrink.

ADTs can be built from other ADTs, not just from arrays: a queue from **two stacks**; a stack from a linked list (push = prepend a head **node** 节点); a queue from a linked list with head and tail **pointers** 指针; a **binary tree** 二叉树 from nodes with two child pointers; a **dictionary** 字典 stores key→value pairs (often on a hash table). Layering this way separates concerns — the algorithm using the ADT need not know how it is built.




*A binary tree: each node has up to two child nodes*




*Three depth-first traversals of a binary tree: pre-order, in-order (sorted order) and post-order*

# Comparing algorithms

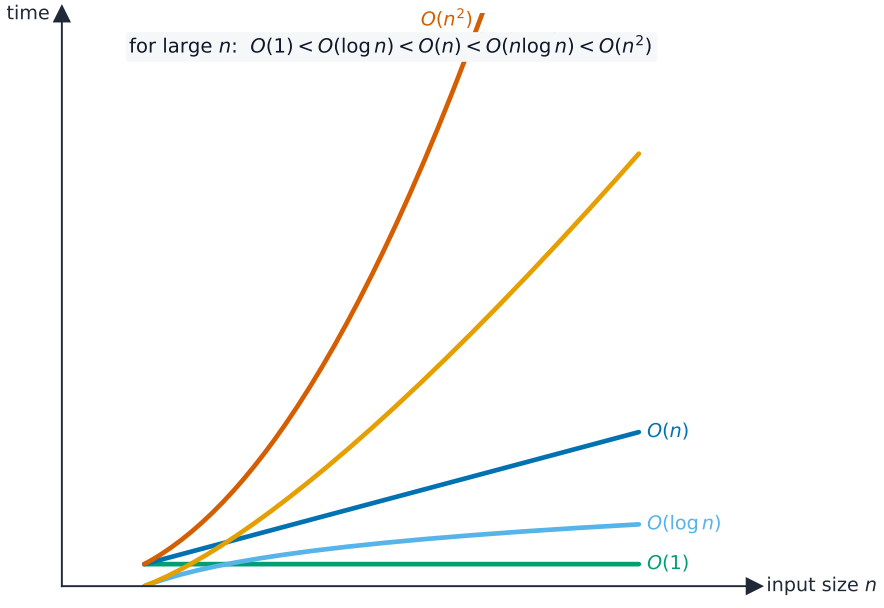
## Time complexity

**Time complexity** 时间复杂度 is how the running time grows with input size $n$, written in **Big-O notation** 大 O 表示法 (the dominant term): $O(1)$ constant, $O(\log n)$ binary search, $O(n)$ linear search, $O(n \log n)$ good sorts, $O(n^2)$ bubble/insertion sort. A smaller order is better at scale, even if another algorithm is faster for small $n$.

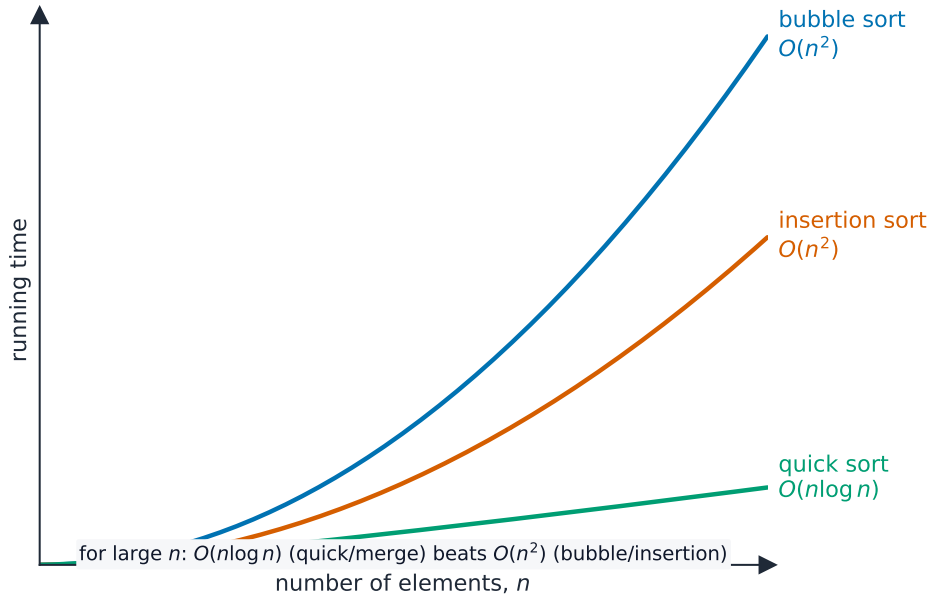
To make that concrete: to sort a million items, an $O(n \log n)$ sort finishes in a fraction of a second, while an $O(n^2)$ sort can take minutes.

**Worked example.** A sorted list holds 1000 items. How many comparisons does each search need in the worst case?

A linear search checks items one at a time, so it may need up to 1000 comparisons — this is $O(n)$. A binary search halves the list each step, so it needs at most $\lceil \log_2 1000 \rceil = 10$ comparisons — this is $O(\log n)$. Doubling the list to 2000 items adds only **one** comparison to the binary search, but up to another 1000 to the linear search — which is why the order of growth, not raw speed, decides the winner at scale.




*How the common orders of growth compare: a smaller order wins at scale*




*How sorting time grows with the number of elements $n$: $O(n^2)$ sorts climb away from an $O(n \log n)$ sort*

### Space complexity

**Space complexity** 空间复杂度 is the extra memory needed. Bubble and insertion sort use O(1) extra (in place); merge sort uses O($n$); recursion uses stack memory proportional to its depth. There is often a time–memory trade-off.

### Other criteria

Simplicity (easier to code and maintain), stability, and adaptiveness (faster on nearly-sorted data). The right algorithm depends on the data and the constraints.

## Recursion

**Recursive algorithms** use **recursion** 递归: the routine calls itself with a smaller version of the same problem, until a **base case** 基本情形 ends the chain. It has two parts: the base case (small enough to solve directly — without it the recursion never stops) and the **recursive case** 递归情形 (reduce the input and call itself).

**Factorial** 阶乘:

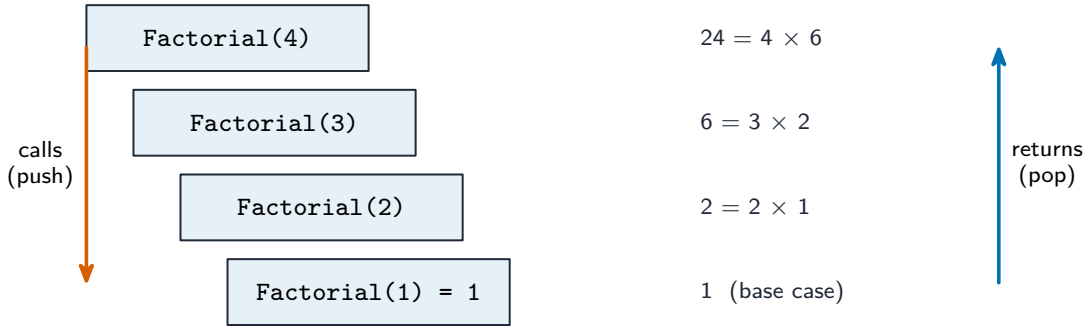
```
FUNCTION Factorial(n : INTEGER) RETURNS INTEGER
    IF n = 0 OR n = 1 THEN
        RETURN 1
```

```
    ELSE
        RETURN n * Factorial(n - 1)
    ENDIF
ENDFUNCTION
```

Recursion is natural for self-similar problems: trees, **divide-and-conquer** 分治 (binary search, merge sort), and nested data. When it is a poor fit, a loop is usually cleaner.

### Tracing a recursive call

For `Factorial(4)`: the calls go down to `Factorial(1)=1`, then **unwinding** multiplies back up: `2*1=2`, `3*2=6`, `4*6=24`. Final result `24`. Track each pending call on a stack.




*Recursion uses the call stack: calls push frames down to the base case, then returns unwind back up*

### Risks

- **infinite recursion** if the base case is missed — crashes with a **stack overflow** 栈溢出.
- high memory use for deep recursion.
- slow if it repeats work (naive Fibonacci is exponential — use a loop or **memoisation** 记忆化).

## What the compiler does for recursive code

Recursion needs **each call to have its own copy** of its **parameters** 参数 and **local variables** 局部变量. The compiler keeps these on the **call stack** 调用栈. For each call it pushes a **stack frame** 栈帧 holding the parameters, the local variables, and the **return address** 返回地址 (where to resume in the caller). When the function

returns, the return value is handed back, the frame is popped, and control resumes at the return address.

Because each call has its own frame, recursive calls don't trample each other's variables. The stack can grow large for deep recursion, which is why very deep recursion may overflow it. This is the same call-and-return mechanism used for ordinary (non-recursive) calls — there is no special "recursion mechanism".

# Exam tips

- Match each algorithm to its **Big-O**: linear search $O(n)$, binary search $O(\log n)$, bubble/insertion $O(n^2)$, good sorts $O(n \log n)$.
- **Binary search needs a sorted list** and halves the search space each step.
- A **recursive** routine needs a base case and a call to itself; explain how deep recursion overflows the call stack.
- Trace a sort or search with a table when asked, showing each pass.

# Further Programming

A-Level Computer Science

## Programming paradigms

A **programming paradigm** 编程范式 is a style of programming — a way of structuring programs, with its own ideas and language features. Four **programming paradigms** are in this syllabus.


*Four paradigms: low-level, imperative, object-oriented and declarative*

### Low-level programming

Programming **close to the hardware** in **machine code** 机器码 or **assembly language** 汇编语言, where each instruction maps to what the CPU runs. It gives direct access to **registers** 寄存器 and **memory addresses** 内存地址, using different **addressing modes** 寻址方式 (immediate, direct, indirect, indexed and relative). It is very fast and compact, but architecture-specific, tedious, and hard to maintain. This is **low-level** 低级 programming, used for device drivers, firmware and bootloaders.

### Imperative (procedural) programming

In **imperative programming** 命令式编程 the programmer writes a **sequence of commands** that change the program's state — assignments, conditionals, loops, function calls. **Variables** 变量 hold state; statements change it; code is organised into procedures and functions (also called **structured** or **structural programming**). This is the style of Topics 9 and 11 (Python, C). Strong when the algorithm has clear sequential steps.

## Object-oriented programming (OOP)

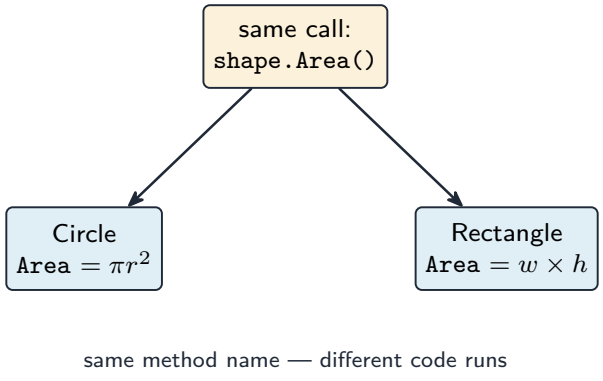
In **object-oriented programming** 面向对象编程 programs are built from **objects** 对象 — units combining **data** (**attributes** 属性) and **operations** (**methods** 方法). Objects are **instances** 实例 of **classes** 类. The four pillars:

- **encapsulation** 封装 — an object's data is hidden behind its methods; outside code uses the public methods only, not the data directly. This protects the object and lets its internals change without breaking callers. For example, a `BankAccount` hides its `balance`; you change it only through `deposit()` and `withdraw()`, which can enforce a rule like "never go below zero".
- **inheritance** 继承 — a **subclass** 子类 specialises a **superclass** 父类, inheriting its attributes and methods and adding or **overriding** 重写 them. Models "is-a" ("a Manager is an Employee").
- **polymorphism** 多态 — different objects respond to the **same method call** differently; the caller need not know the exact type. Every `Shape` has `Area()`, and a `Circle` and a `Rectangle` each implement it their own way.
- **abstraction** 抽象 — show a simple interface and hide the implementation.

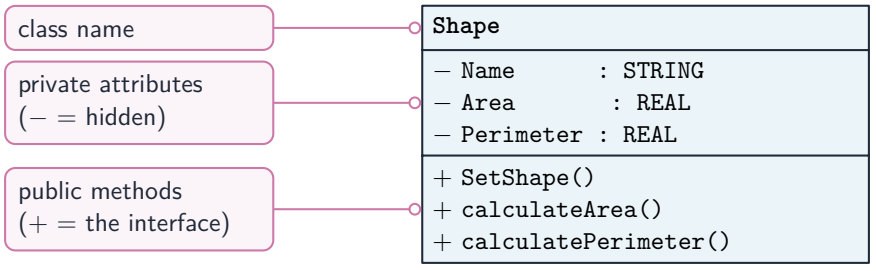
Other terms:

- a **constructor** 构造函数 is a special method run when an object is created, to set up its attributes.
- **getters** and **setters** read and write an object's attributes (its **properties**) through methods.
- **aggregation** 聚合 and **containment** 包含 build an object from other objects (a "has-a" relationship).

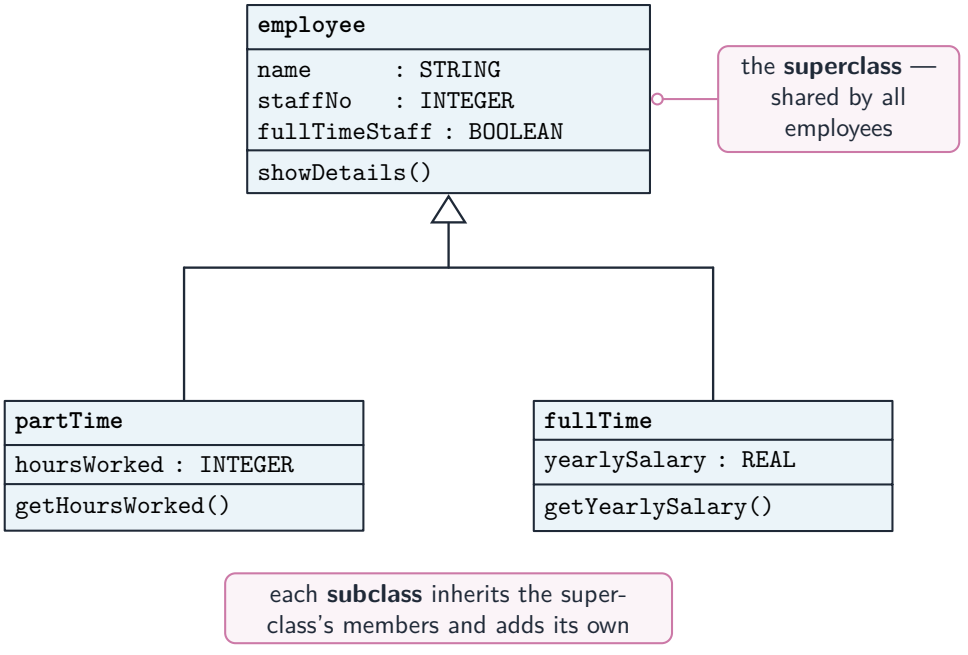
OOP is used for large systems, GUIs, simulations and games.



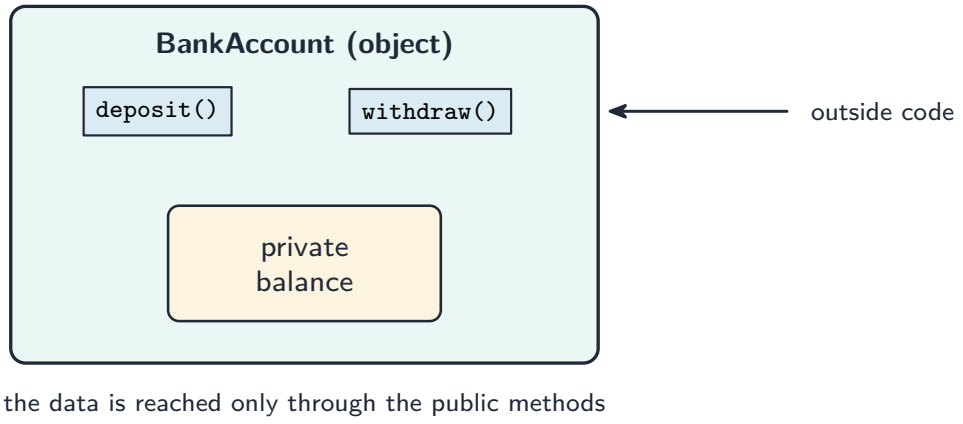

*Polymorphism: the same method call runs each object's own code*




*A class diagram for a Shape: private attributes and public methods*




*Inheritance: partTime and fullTime are subclasses of employee*




*Encapsulation: an object's data is private, reached only through its public methods*

## Declarative programming

In **declarative programming** 声明式编程 you say **what** to compute, not **how** — the runtime works out the steps. Two kinds:

- **functional programming** 函数式编程 — built from **pure functions** 纯函数 (no **side effects** 副作用; same input always gives the same output) composed together. Examples: Haskell, Lisp.
- **logic programming** 逻辑编程 — state facts and rules; the engine answers a **goal** (query) by inference. Example: Prolog.

A familiar declarative example is **SQL** 结构化查询语言: `SELECT * FROM Customer WHERE Country = 'UK'` says what you want, not how to walk the records.

## Comparing paradigms

| Paradigm | Strength | Typical languages |
|---|---|---|
| Low-level | maximum control, speed | assembly |
| Imperative | direct, intuitive | C, Python |
| Object-oriented | modular, models entities | Java, C#, Python |
| Functional | clear, no side effects | Haskell, F# |
| Logic | inference, rules | Prolog |
| Database | data queries | SQL |

Modern languages often **mix paradigms** — Python supports all of procedural, OOP and functional. The right one depends on the problem.

# File processing

This extends the **file** 文件 handling from Topic 10, processing **serial**, **sequential** and **random** (direct-access) files. Pseudocode operations: `OPENFILE name FOR READ | WRITE | APPEND` (READ opens an existing file, WRITE creates/overwrites, APPEND adds to the end); `READFILE name, line`; `WRITEFILE name, value`; `CLOSEFILE name`; and `EOF(name)` which is TRUE at the end.

Read a whole file:

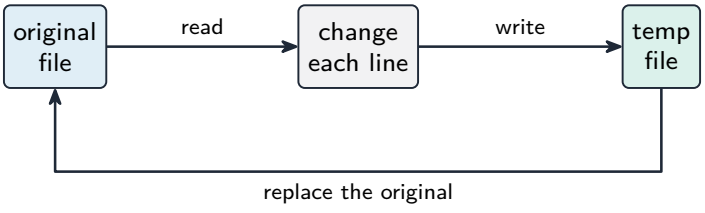
```
OPENFILE "names.txt" FOR READ
WHILE NOT EOF("names.txt") DO
    READFILE "names.txt", thisName
    OUTPUT thisName
ENDWHILE
CLOSEFILE "names.txt"
```

Search a file (stop when found):

```
found ← FALSE
OPENFILE "people.txt" FOR READ
WHILE NOT EOF("people.txt") AND NOT found DO
    READFILE "people.txt", line
    IF line = target THEN found ← TRUE
ENDWHILE
CLOSEFILE "people.txt"
```

## Updating a file in place

Most languages can't edit a text file in place. Instead: open the original for READ and a temporary file for WRITE; for each line, write the new version if it should change, else the original; close both; then replace the original with the temp file. The same pattern handles deleting lines (skip them) and inserting lines.




*Updating a file in place: read the original, write changes to a temp file, then replace the original*

### Pitfalls

Forgetting to close a file (data may be lost); opening for WRITE when you meant APPEND (overwrites everything); reading past `EOF`; hard-coded paths — a path like `/Users/Admin/data.txt` breaks on another machine, so use a relative constant such as `DataFile = "./data/scores.txt"`.

## Exception handling

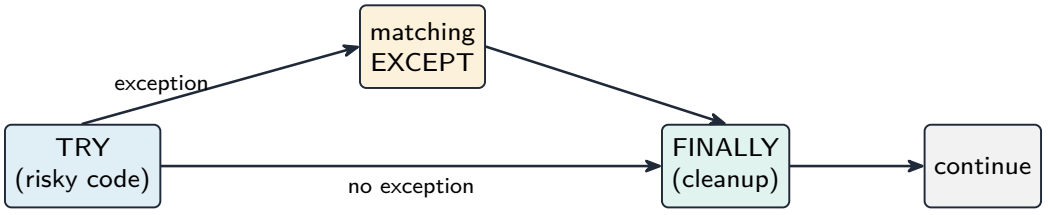
An **exception** 异常 is an error or unexpected condition during execution — divide by zero, file not found, network failure, an **array** 数组 index out of range. **Exception handling** 异常处理 lets a program **detect** it and respond gracefully instead of crashing.

It matters because real programs face errors that **cannot be prevented up front** (files moved, networks down, bad input); without it, every operation needs its own `IF` check; and it **separates** the normal flow from the error handling, so the main path reads cleanly. For example, a file may be deleted by another user between your program checking it exists and actually opening it — you cannot prevent that, only handle the failure when it happens.

## Pattern

```
TRY
    OPENFILE "data.txt" FOR READ
    READFILE "data.txt", line
    OUTPUT line
    CLOSEFILE "data.txt"
EXCEPT FileNotFound
    OUTPUT "Sorry, the file does not exist."
EXCEPT ReadError
    OUTPUT "Sorry, error reading the file."
ENDTRY
```

The `TRY` block holds the code that might fail; the first matching `EXCEPT` block runs. Real languages also have a catch-all `EXCEPT` and a `FINALLY` block that runs whether or not an exception happened — useful for cleanup (closing files).




*Exception flow: an exception jumps to the matching EXCEPT; FINALLY always runs before the program continues*

## Raising an exception

A subroutine that detects an error can **raise** 抛出 an exception so the caller handles it:

```
PROCEDURE Divide(a : INTEGER, b : INTEGER) RETURNS INTEGER
    IF b = 0 THEN
        RAISE DivideByZero
    ENDIF
    RETURN a DIV b
ENDPROCEDURE
```

### Where to handle exceptions

Handle them **close to the error** if the response is simple (a message, a retry), or **higher up the call stack** 调用栈 if only the outer code knows what to do (a top-level GUI loop logs the error and shows a friendly dialog). **Don't swallow exceptions silently** — at least log them, or debugging becomes impossible.

Common exceptions: `FileNotFound`, `IOError`, `DivisionByZero`, `IndexOutOfRange`, `InvalidArgument`, `NullReference`, `OutOfMemory`. Wrapping each failing operation in a `TRY` with the right `EXCEPT` handlers gives a program that **degrades gracefully** instead of crashing.

**Worked example.** A text file of members needs one member's phone number changed. Why can the program not simply overwrite that line, and what is the pattern? A text file's lines are **different lengths**, and the file has no gaps to absorb a difference: a longer replacement would run into the next record, and a shorter one would leave part of the old line behind. So the pattern is to open the original for **READ** and a **temporary** file for **WRITE**, read every line in turn, writing the **new** version for the line that changes and the **original** line for all the others, close both, then **replace** the original with the temporary file. The same shape handles deleting (skip the line) and inserting (write the extra line). Note that **every** line gets written, not only the changed one - writing just the new record and losing the rest of the file is the classic slip.

## Exam tips

- Distinguish the **paradigms** (procedural, object-oriented, declarative, low-level) and when each suits a problem.
- For OOP, define **class, object, inheritance, encapsulation and polymorphism** with a short example.
- Explain **exception handling** (try/catch) and why it beats letting the program crash.