1. What is a program and how does it function?

A *program* is a set of ordered instructions written in a programming language that tells a computer what actions to perform. When you run a program the computer's processor and operating system load the program into memory, read its instructions one-by-one, and execute them. Execution may include arithmetic and logical operations, reading/writing data to memory or disk, interacting with input/output devices, and calling other programs or library routines. A program usually processes input, changes state (stores results), and produces output; errors or exceptions occur when an instruction cannot be completed.

2. What are the key steps involved in the programming process?

Typical steps are:

1. **Problem understanding / requirements:** Know exactly what needs to be solved.
2. **Algorithm design / planning:** Create a step-by-step approach (flowchart, pseudocode).
3. **Coding / implementation:** Translate the algorithm into source code using a programming language.
4. **Compilation / interpretation:** Convert source code to machine-executable form (compile or interpret).
5. **Testing & debugging:** Run the program with test inputs, find and fix errors.
6. **Documentation:** Write comments and external docs so others (or you later) can understand the code.
7. **Deployment / maintenance:** Release the program to users and update it when needed.
   These steps often repeat (iterative) — especially testing and maintenance — until the software meets requirements.

3. Main differences between high-level and low-level programming languages

- **Abstraction:** High-level languages (Python, Java, C#) provide strong abstraction from hardware — easier syntax, data types and library functions. Low-level languages (assembly, machine code) map closely to CPU instructions.
- **Readability & productivity:** High-level code is easier to read/write and faster to develop; low-level code is harder and time-consuming.

- **Control & performance:** Low-level gives more direct control over hardware and can be more efficient in time/space; high-level may introduce runtime overhead but is usually fast enough and more maintainable.
- **Portability:** High-level languages are more portable across platforms; low-level code is typically tied to a specific processor/architecture.
- **Use cases:** High-level for application development, scripts and business logic; low-level for device drivers, embedded systems, and critical performance hotspots.

4. Describe the roles of the client and server in web communication

- **Client:** The client is usually a user's browser or app that initiates requests (e.g., HTTP GET) to access resources or services. It presents the user interface, collects input, sends requests, and renders responses (HTML, JSON, images).
- **Server:** The server waits for client requests, processes them (runs business logic, queries databases), and returns responses. Servers host resources, enforce security, and often coordinate backend services (databases, authentication, caching). Together they follow request–response interactions: client requests → server processes → server responds.

5. Network layers on client and server & Explain client–server communication

Modern networked applications use layered models (like the OSI or TCP/IP stacks). Typical layers relevant to client/server:

- **Application layer:** HTTP/HTTPS, SMTP, FTP — client generates application requests; server handles them.
- **Transport layer:** TCP or UDP — ensures reliable (TCP) or fast (UDP) delivery between endpoints.
- **Network layer:** IP — routes packets across networks.
- **Link/Physical layers:** Ethernet/Wi-Fi and physical transmission.

Client–server communication flow (example HTTP over TCP/IP):

1. Client resolves server address (DNS).
2. Client creates TCP connection to server IP:port (three-way handshake).
3. Client sends HTTP request over TCP.
4. Server processes request, interacts with backend, and forms HTTP response.

5. Server sends response back; client receives and renders it.
6. Connection closes or is reused (keep-alive).
   This layered approach isolates concerns (routing, reliability, application logic) and enables interoperability.

6. How does broadband differ from fiber-optic internet?

- **Broadband** is a generic term for high-speed internet access that is always-on; it includes DSL, cable, satellite, and fiber.
- **Fiber-optic** is a specific broadband technology that uses glass fibers to transmit data as light pulses. Compared to many other broadband types, fiber typically offers: much higher bandwidth (Gbps-level speeds possible), lower latency, and better reliability and signal quality over long distances. Other broadband types (DSL, cable) use electrical signals on copper or coax and often have lower peak speeds and higher latency. Fiber is considered the highest-performance broadband for symmetric upload/download and future-proof capacity.

7. Differences between HTTP and HTTPS protocols

- **HTTP (Hypertext Transfer Protocol):** Transmits data in plaintext and does not provide confidentiality, integrity, or authentication.
- **HTTPS:** HTTP over TLS/SSL — provides encryption (confidentiality), message integrity, and server (and optionally client) authentication via certificates. HTTPS prevents eavesdropping and tampering and is required for transmitting sensitive data (passwords, payment). Functionally, they are the same application protocol, but HTTPS runs inside an encrypted channel created by TLS.

8. What is the role of encryption in securing applications?

Encryption converts readable data into an unreadable form (ciphertext) that only authorized parties with the correct keys can decode. Roles include:

- **Confidentiality:** Prevents eavesdroppers from reading sensitive data in transit (TLS) or at rest (disk/file encryption).
- **Integrity:** Cryptographic checks (MACs, HMACs) detect tampering.
- **Authentication & non-repudiation:** Digital signatures and certificates verify who sent data and prevent denial.

Encryption is applied at multiple layers (transport, database, file system) and is a key control in protecting user privacy and meeting compliance requirements.

9. Difference between system software and application software

- **System software:** Low-level software that manages hardware and provides a platform for running applications (e.g., operating systems like Windows/Linux, device drivers, utilities, system libraries).
- **Application software:** Programs that perform specific user-facing tasks (e.g., word processors, web browsers, accounting software).
  In short: system software runs the computer and supports resources; application software uses those resources to do useful work for users.

10. Significance of modularity in software architecture

Modularity divides a system into separate components or modules with well-defined interfaces. Benefits:

- **Maintainability:** Smaller modules are easier to understand, debug, and update.
- **Reusability:** Modules can be reused across projects.
- **Isolation:** Bugs and changes are localized, reducing ripple effects.
- **Parallel development:** Teams can work on different modules simultaneously.
- **Testability:** Modules can be unit-tested independently. Overall, modularity promotes cleaner design and reduces long-term costs.

11. Why are layers important in software architecture?

Layering separates concerns into horizontal strata (e.g., presentation, business logic, data access). Importance:

- **Separation of concerns:** Each layer handles specific responsibilities.
- **Replaceability:** One layer can be swapped with minimal impact on others.
- **Scalability & deployment flexibility:** Layers can be distributed across servers or scaled independently.
- **Security:** Layers allow enforcing controls at specific boundaries.
  Layers improve clarity, manageability, and make large systems easier to evolve.

12. Explain the importance of a development environment in software production

A development environment (IDE, tools, local server, debuggers, build system) increases developer productivity and code quality. It provides: syntax highlighting, auto-completion, build/test automation, debugging, version control integration, and local deployment. A consistent environment (via containers or VMs) reduces “it works on my machine” issues and speeds onboarding. Good environments accelerate iteration and ensure reproducible builds.

13. Difference between source code and machine code

- **Source code:** Human-readable instructions written in a programming language (e.g., C, Python). It is understandable and editable by developers.
- **Machine code:** Binary instructions executed directly by the CPU (0s and 1s). Compilers or interpreters translate source code into machine code (or bytecode) so the machine can perform operations. Source code is for humans; machine code is for the processor.

14. Why is version control important in software development?

Version control (e.g., Git) tracks changes to code over time. Benefits:

- **History & rollback:** You can see who changed what and revert to earlier versions.
- **Branching & merging:** Experiment and develop features in isolation before merging.
- **Collaboration:** Multiple developers can work concurrently without overwriting each other.
- **Traceability:** Link commits to issues, reviews, and releases.
  Overall, it enables safe teamwork and preserves project history.

15. What are the benefits of using GitHub for students?

- **Portfolio & visibility:** Host projects publicly to show to employers/teachers.
- **Version control learning:** Practical experience with Git workflows.
- **Collaboration:** Work on group assignments via pull requests and issue tracking.
- **Free Education Resources:** GitHub Student Pack provides free tools/services (CI, cloud credits, IDEs).

- **Community & open-source contribution:** Join open-source projects and learn from real codebases.

16. Differences between open-source and proprietary software

- **Source availability:** Open-source provides source code for inspection, modification, and redistribution; proprietary keeps source code closed.
- **License & freedoms:** Open-source licenses permit reuse and modification (with conditions), proprietary licenses restrict usage and redistribution.
- **Community vs vendor:** Open-source often has community-driven development; proprietary is developed and supported by a vendor.
- **Cost & support model:** Open-source can be free (or commercial support can be paid); proprietary usually requires purchase or subscription.
- **Innovation & control:** Open-source allows customization; proprietary can offer tightly controlled, possibly more polished user experience and guaranteed vendor support.

17. How does Git improve collaboration in a software development team?

Git enables branching and merging, allowing each developer to work independently on features or fixes. Pull requests and code reviews facilitate discussion and quality checks before integration. The commit history provides accountability and a clear narrative of changes. Conflicts are manageable and traceable, and tools built on Git (CI/CD) automate testing and integration, reducing integration pain and improving release confidence.

18. Role of application software in businesses

Application software automates business processes, increases productivity, and enables decision-making. Examples: ERP for resource planning, CRM for customer relationships, accounting software for finance, and BI tools for analytics. Applications streamline workflows, reduce manual errors, and provide data for strategic decisions. They often integrate with databases and other systems, becoming central assets for business operations.

19. Main stages of the software development process

Common staged model (waterfall) or iterative models (Agile) but core stages are:

1. **Requirement analysis** — understand and document needs.
2. **Design** — architecture and detailed design (system/DB/UI).
3. **Implementation** — coding the solution.
4. **Testing** — verify correctness, performance, security.
5. **Deployment** — release to production.
6. **Maintenance** — updates, bug fixes, enhancements.
   Agile approaches iterate these stages in short cycles (sprints).

20. Why is the requirement analysis phase critical?

Requirement analysis establishes what the system must do; mistakes here lead to building the wrong product. It reduces scope creep, clarifies stakeholder expectations, enables accurate cost/time estimates, and forms the baseline for design and testing. Clear requirements help prevent rework and save time and money.

21. What is the role of software analysis in the development process?

Software analysis breaks down requirements into detailed functional and non-functional specifications, models (DFDs, ER diagrams), and identifies constraints. It informs the system design by revealing data flows, use cases, performance needs, and integration points. Analysis ensures design decisions are grounded in business needs and helps estimate effort and risks.

22. What are the key elements of system design?

Key elements include:

- **Architecture style:** e.g., layered, microservices, monolith.
- **Modules & interfaces:** Define components and their interaction points.
- **Data design:** Database schema, data models, persistence strategy.
- **API design & contracts:** How components communicate (REST, gRPC).
- **Non-functional design:** Scalability, performance, security, availability, and fault tolerance.

- **Deployment & ops considerations:** How it will be hosted, CI/CD, monitoring. Good system design balances functional requirements with these cross-cutting concerns.

23. Why is software testing important?

Testing verifies that software meets functional and non-functional requirements, finds defects before release, improves reliability, and builds user trust. Types of testing (unit, integration, system, acceptance) catch different kinds of issues. Effective testing reduces maintenance costs and security risks while ensuring the product behaves correctly under real-world conditions.

24. What types of software maintenance are there?

- **Corrective maintenance:** Fixing bugs found after release.
- **Adaptive maintenance:** Updating software for new environments (OS upgrades, hardware changes).
- **Perfective maintenance:** Enhancements and performance improvements based on user feedback.
- **Preventive maintenance:** Changes to prevent future problems (refactoring, code cleanup).
  Each type aims to extend the useful life and relevance of the software.

25. Key differences between web and desktop applications

- **Delivery & installation:** Desktop apps are installed on a user's machine; web apps run in browsers and need no local install.
- **Platform dependence:** Desktop apps can be platform-specific (Windows/Linux/Mac); web apps are largely platform-independent.
- **Connectivity:** Web apps usually require network access; desktop apps can work offline (depending).
- **Deployment & updates:** Web apps are updated centrally by the server; desktop apps require update mechanisms on each client.
- **Performance & capabilities:** Desktop apps can access local hardware more directly and may offer better performance for heavy tasks; web apps are constrained by browser APIs but are improving.

26. Advantages of using web applications over desktop applications

- **Ease of access:** Run from any device with a browser.
- **Centralized updates:** Immediate rollout of fixes/features.
- **Cross-platform compatibility:** One codebase reaches many systems.
- **Lower client requirements:** Minimal installation, easier onboarding.
- **Scalability:** Server-side scaling supports many users; simpler backup and data centralization.
  These benefits make web apps ideal for distributed teams and frequent updates.

27. Role of UI/UX design in application development

UI/UX determines how users perceive and interact with software. Good design improves usability, reduces errors, and increases satisfaction and productivity. UX focuses on the user journey and flow; UI handles visual and interactive elements. Investing in UI/UX reduces support costs, increases adoption, and directly impacts the success of the application.

28. Differences between native and hybrid mobile apps

- **Native apps:** Built for a specific platform using platform languages/frameworks (Swift for iOS, Kotlin/Java for Android). They have full access to device features, typically better performance and UX.
- **Hybrid apps:** Built using web technologies (HTML/CSS/JS) and run inside a native wrapper (e.g., Cordova, Ionic, React Native to varying degrees). They provide cross-platform code reuse and faster development but may have limitations in performance or native feature access (though modern frameworks reduce this gap). Choice depends on performance needs, time-to-market, and development resources.

29. Significance of Data Flow Diagrams (DFDs) in system analysis

DFDs graphically represent the flow of data through systems, showing processes, data stores, external entities, and data movement. They help analysts understand system boundaries, inputs/outputs, and how information transforms. DFDs are useful for communicating requirements with stakeholders and identifying redundancies or inefficiencies before coding begins.

30. Pros and cons of desktop applications compared to web applications

Pros of desktop apps:

- Better performance for resource-heavy tasks.
- Full access to hardware and OS features.
- Often work offline without network dependency.

Cons of desktop apps:

- Harder to distribute and update (per-machine installs).
- Platform-specific development effort.
- Less convenient remote access and centralized data handling.

Pros of web apps: (see Q26) centralized updates, cross-platform, easy access.
Cons of web apps:

- Dependence on network connectivity and browser capabilities.
- Potentially limited access to device hardware and lower raw performance for certain tasks.
  Choosing depends on functional needs and user context.

31. How do flowcharts help in programming and system design?

Flowcharts visually map control flow and decision points of an algorithm or process. They make logic easier to reason about, help spot logical errors early, and serve as documentation for developers and stakeholders. Flowcharts are a helpful step before writing code because they provide a clear, language-neutral representation of the program's structure.