**Server & Infrastructure**

1. SQL vs NOSQL:

a. SQL databases represent data in form of tables which consists of n number of rows of data whereas NoSQL databases are the collection of key-value pair, documents, graph databases or wide-column stores which do not have standard schema definitions which it needs to adhered to.

b. SQL databases have predefined schema whereas NoSQL databases have dynamic schema for unstructured data.

c. In most typical situations, SQL databases are vertically scalable. You can manage increasing load by increasing the CPU, RAM, SSD, etc, on a single server. On the other hand, NoSQL databases are horizontally scalable. You can just add few more servers easily in your NoSQL database infrastructure to handle the large traffic.

d. For complex queries: SQL databases are good fit for the complex query intensive environment whereas NoSQL databases are not good fit for complex queries. On a high-level, NoSQL don’t have standard interfaces to perform complex queries, and the queries themselves in NoSQL are not as powerful as SQL query language.

e. SQL databases are not best fit for hierarchical data storage. But, NoSQL database fits better for the hierarchical data storage as it follows the key-value pair way of storing data similar to JSON data. NoSQL database are highly preferred for large data set (i.e for big data). Hbase is an example for this purpose.

f. SQL databases are best fit for heavy duty transactional type applications, as it is more stable and promises the atomicity as well as integrity of the data. While you can use NoSQL for transactions purpose, it is still not comparable and sable enough in high load and for complex transactional applications.

g. SQL databases emphasizes on ACID properties ( Atomicity, Consistency, Isolation and Durability) whereas the NoSQL database follows the Brewers CAP theorem ( Consistency, Availability and Partition tolerance )

2. Which Database Is Right For Your Business?

MySQL is a strong choice for any business that will benefit from its pre-defined structure and set schemas. For example, applications that require multi-row transactions - like accounting systems or systems that monitor inventory - or that run on legacy systems will thrive with the MySQL structure.

MongoDB, on the other hand, is a good choice for businesses that have rapid growth or databases with no clear schema definitions. More specifically, if you cannot define a schema for your database, if you find yourself denormalizing data schemas, or if your schema continues to change - as is often the case with mobile apps, real-time analytics, content management systems, etc.- MongoDB can be a strong choice for you.

When strong consistency is not a problem, you can choose NoSQL. Note that NOSQL will obviously have eventual consistency.

3. Redis DB: It is consider as one of the fastest NoSQL server as it works with the in-memory dataset.

4. HTTP vs HTTPS:

Hyper Text Transfer Protocol Secure (HTTPS) is the secure version of HTTP, the protocol over which data is sent between your browser and the website that you are connected to. The 'S' at the end of HTTPS stands for 'Secure'. It means all communications between your browser and the website are encrypted. HTTPS is often used to protect highly confidential online transactions like online banking and online shopping order forms.

How does HTTPS work?

HTTPS pages typically use one of two secure protocols to encrypt communications - SSL (Secure Sockets Layer) or TLS (Transport Layer Security). Both the TLS and SSL protocols use what is known as an 'asymmetric' Public Key Infrastructure (PKI) system. An asymmetric system uses two 'keys' to encrypt communications, a 'public' key and a 'private' key. Anything encrypted with the public key can only be decrypted by the private key and vice-versa.

As the names suggest, the 'private' key should be kept strictly protected and should only be accessible the owner of the private key. In the case of a website, the private key remains securely ensconced on the web server. Conversely, the public key is intended to be distributed to anybody and everybody that needs to be able to decrypt information that was encrypted with the private key.

What is a HTTPS certificate?

When you request a HTTPS connection to a webpage, the website will initially send its SSL certificate to your browser. This certificate contains the public key needed to begin the secure session. Based on this initial exchange, your browser and the website then initiate the 'SSL handshake'. The SSL handshake involves the generation of shared secrets to establish a uniquely secure connection between yourself and the website.

When a trusted SSL Digital Certificate is used during a HTTPS connection, users will see a padlock icon in the browser address bar. When an Extended Validation Certificate is installed on a web site, the address bar will turn green.

Why is a SSL certificate required?

All communications sent over regular HTTP connections are in 'plain text' and can be read by any hacker that manages to break into the connection between your browser and the website. This presents a clear danger if the 'communication' is on an order form and includes your credit card details or social security number. With a HTTPS connection, all communications are securely encrypted. This means that even if somebody managed to break into the connection, they would not be able decrypt any of the data which passes between you and the website. These attacks are called 'Man in the Middle' attacks.

5. What is reverse proxy?

The server that retrieves resources on behalf of a client from one or more servers. These resources are then returned to the client appearing as if they originated from the proxy server itself.

6. What is the difference between Web Server/HTTP Server and Application Server?

Application Server is sometimes mixed up with web server while a web server handles mainly HTTP protocol and the application server deals with several different protocols including, but not limited to HTTP.

In case of AutoMeta, nginx is the webserver and thin is the application server. nginx will take the HTTP request and then give it to the application server for processing where the complete business logic will be implemented and the result is displayed on the browser. Application server can also handle HTTP protocols directly without the need of nginx. The benefit of nginx is that it will redirect my request to different application server instances (:4000, :4001, ....). If application server had to handle this, then the users would have to type the ports of the application server instances directly: www.xyz.com:4008

6. What is NGINX?

NGINX is a HTTP server (web server), reverse proxy, caching, load balancing, media streaming and more. The default port for nginx is 80. Apache is another web server.

6. What is HAProxy?

High Availability Proxy provides a high available load balancer and proxy server for TCP/HTTP based application. It is written in C and has a reputation for being fast and efficient in terms of processor and memory usage. Default port is 8100.

7. What is SSL?

Secure Socket Layer which is a standard security protocol for establishing encrypted links between a web server and a browser in an online communication.

The browser/server checks to see whether or not it trusts the SSL certificate and it if does, it sends a message to the browser. The web browser sends back a digitally signed acknowledgement to start an SSL encrypted session. Encrypted data is shared between browser and the web server.

8. RabbitMQ

This is a messaging queue mechanism where there can be multiple 'exchanges' and each exchange can have multiple queues. For each action in the controller, we have to create a new queue in one of the exchanges. We are using the exchange name as 'sneakers' in autometa. It has 7 queues in int for each of the seven actions which require background processing of jobs. Default port is 5672

9. Sneakers

This is a background job executor gem in rails. The action which we want to execute in the background should have the exchange and queue configured Example: (queue: abc of exchange 'sneakers'). Now, we would have different files for each worker in the app/worker/ directory. Each file would say that I belong to the queue: "some queue name" and exchange: "some exchange". So, the file which would have written queue: "abc" and exchange: "sneakers" would be picked up and then the action "work" would be executed in the same file.

10. Sharding

Sharding is a type of database partitioning that separates very large databases into smaller, faster and more easily managed parts called data shards.

How it works?

It creates a map table that contains information about the range of primary keys of different table in different shards. Whenever we get a query based on the ID in query, shard can be identified using the map table.

It is of two types:

a. Horizontal Sharding: When my DB table/collection is having a lot of rows or growing vertically, then we do Horizontal Sharding.

Pros:

1. A new database can be added easily. Just make a new database and add the entry in the map table. That's it.

2. Very obviously, the query becomes very fast. As the size of shard is small as compared to original DB.

Cons:

1. Aggregation queries like MAX, MIN, AVG, SUM are not supported. As we need to traverse to each of the shard to get the aggregated value.

b. Vertical Sharding: When my DB table/collection is having a lot of columns out of which some columns are required very rarely, then I can shard them vertically to store them in a different database. Note that both the tables now would have the primary key.

Pros:

1. Aggregation queries are supported easily (Min, Max, Avg, Sum of a specific column) Cons:

1. It becomes difficult to support a new query which requires columns from both the shards.

2. Extra space is required to keep the primary key across shards.

11. Difference between Partitioning and Sharding

The difference between Partitioning and Sharding is mostly semantic. With Partitioning, your tables will still be in the same database instance whereas with Sharding, your tables are split across multiple database servers.

12. Scaling

Scaling of an application is termed as increasing the capacity to handle multiple request without compromising the availability of the application.

If incoming request is huge, then the application server has ability to handle limited number of requests, so we have to scale the application server in some way so that it is able to handle more requests.

It's of two types:

a. Horizontal: Adding new components (application server, DB, etc) to the application. For example, We take a new machine and start another application server there. We now need a load balancer on top of both the machines so that the load is distributed across these machines/servers.

Pros:

1. Highly scalable. As new component can be added at any point of time.

2. New components can be added incrementally. No downtime required.

3. Comparatively cheaper. 10 components of computation power x will be cheaper as compared to single component with 10x computation power.

4. No single point of failure.

5. We can serve request from all over the globe in good response time by keeping servers in different geographical locations.

Cons:

1. Synchronization b/w different components need to be handled carefully.

2. Communication delay or synchronization delay is there.

b. Vertical: adding extra capability to existing components in the application.

Pros:

1. We have only one component with very high computation. So there if no synchronization required.

2. No communication delay.

Cons:

1. Very expensive.

2. Single point of failure.

3. Not so easy to scale. Migration is very difficult: Downtime is required.

4. Since I only have one server, request coming far away around the globe would be slow.

13. Why did you choose MongoDB in AutoMeta and not MySQL or anything?

We had to choose any two from Consistency, Availability and Partitioning (as per CAP theorem).

We chose:

1.Consistency: It is not something that is of utmost importance to us since it will obviously have eventual consistency.

2. Availability: we chose to have a highly available system. We need faster query time. Our data model is such that if we keep too much normalized form of data, it will need so many joins for most of the queries. So we it was a better solution to keep redundant data using MongoDB instead of keeping normalized data in SQL. In MongoDB we keep the copy of data wherever needed.

3. Partitioning: Mongo supports Sharding. So, this is a Yes.

14. Distributed System: A group of computers that work together to achieve some common goal. Lets say you have a website that runs a small service which converts doc to pdf. Gradually, as months passes, your website becomes popular and the server response become slower since it was not made to handle such huge amount of requests.

a. What you can do now is upgrade your existing hardware, better RAM, better processor, etc. This is called Vertical Scaling but how long will you be able to do this?

b. Instead of making your existing hardware better, buy more of similar hardware. So, instead of one machine's upgrade from 64Gb to 128GB, buy another 64GB RAM so that you have two 64GB RAMs. This is called Horizontal Scaling.

When we do point b (Horizontal Scaling), then our task is divided across multiple servers. These servers/set of machines can together be called as Distributed Systems. So, whenever we talk about Distributed Systems, think of it as a system which is scaled horizontally.

15. Distributed Data Stores / Replication in DB : We can replicate the DB server with multiple DB servers so that load is distributed across them.

a. Master-Slave Strategy: We will have multiple database servers in which there will be one master and different slave database servers. Any write operation would happen in the master database. The read operations would happen from the slave databases. The synchronization between the slaves and master after master writes something new happens asynchronously. Since one may access data from slaves before this sync, the data in slaves for this delta time is inconsistent.

The disadvantage of this is the fact that: For larger DBs, the sync delay between master-slave is there.

b. Sharding: Written above (10)

16. CAP Theorem: Consistency - Availability - Partitioning

- Consistency: Database is consistent if at all point of time, it returns the correct value. In the master slave strategy, there is a moment when the master should sync with the slaves and for that time, the data in slaves is inconsistent. So, if a DB adopt this, then the data is inconsistent and can not be trusted all the time.

- Availability: Even if some machine in my cluster goes down, the system should be available up and running.

- Partitioning: If my connection between two machines goes down in a cluster, we should still be able to read and write data. However, the read-write may not be correct (if Availability is chosen over consistency) but still the system should respond if P is chosen.

In reality, one can't have all these three properties in the database. You can either have any of these two combinations. The database give us the configuration in our hands so that we can choose either of the two from these. Even in case of internal network failure, the guarantees (either C or A) promised by system will continue to hold.

Traditional Relational databases like MySQL choose Consistency over Availability.

NoSQL systems believe in A and P by sacrificing C. But remember that we won't have strict consistency but we will be eventual consistency here. Most of the applications can use this where the exact data is not important at all point of time. We have BASE property here meaning Basically Available and SoftState Eventual Consistency.

IRCTC, Banking systems, etc where the data has to be correct at all point of time should use 'RDBMS' which ensures consistency over everything.

17. My website is slow. How should I proceed with this problem?

- Optimize my query:

Maybe by solving n+1 problem. If certain frequently accessed columns are not indexed, we have to index them for faster access. If needed, in the worst case, we can tweak with the DB schema as well.

- Replication/Sharding DB Server:

If DB queries are huge, then we can replicate the DB server with multiple DB servers so that load is distributed across them. We will have to handle the sync delay between master-slave if Master-slave strategy is chose.

- Scaling:

If incoming request is huge, then the application server has ability to handle limited number of requests, so we have to scale the application server in some way so that it is able to handle more requests. It can either be vertical scaling or horizontal scaling.

- Check for Async Workers:

If the job can be done in an async way, then the thread must not wait for the response and can close early. For cases like Sending Notifications, Analytics, this can be used. It can easily be used with help of Kafka or any queue where the request are put and the thread gets a response like "Your request is accepted and will be done in some time..."

18. Monolithic Design vs. Microservices Design

The following may look large to you but when you read it, it's so simple and interesting!

Disadvantages of Monolithic design:

a. Huge Code base: Successful applications have a habit of growing over time and eventually becoming huge. During each sprint, your development team implements a few more user stories, which, of course, means adding many lines of code. After a few years, your small, simple application will have grown into a monstrous monolith

b. Complexity: One major problem is that the application is overwhelmingly complex. It’s simply too large for any single developer to fully understand. As a result, fixing bugs and implementing new features correctly becomes difficult and time consuming.

c. Startup time: The larger the application, the longer the start-up time is.

d. Deployment: Another problem with a large, complex monolithic application is that it is an obstacle to continuous deployment. Today, the state of the art for SaaS applications is to push changes into production many times a day. This is extremely difficult to do with a complex monolith, since you must redeploy the entire application in order to update any one part of it.

e. Scaling: Monolithic applications can also be difficult to scale when different modules have conflicting resource requirements. For example, one module might implement CPU-intensive image processing logic and would ideally be deployed in Amazon EC2 Compute Optimized instances. Another module might be an in-memory database and best suited for EC2 Memory-optimized instances. However, because these modules are deployed together, you have to compromise on the choice of hardware.

f. Reliability: Another problem with monolithic applications is reliability. Because all modules are running within the same process, a bug in any module, such as a memory leak, can potentially bring down the entire process. Moreover, since all instances of the application are identical, that bug will impact the availability of the entire application.

g. Constant Tech Stack: Last but not least, monolithic applications make it extremely difficult to adopt new frameworks and languages. For example, let’s imagine that you have 2 million lines of code written using the XYZ framework. It would be extremely expensive (in both time and cost) to rewrite the entire application to use the newer ABC framework, even if that framework was considerably better. As a result, there is a huge barrier to adopting new technologies. You are stuck with whatever technology choices you made at the start of the project.

Advantages of Microservices design:

a. First, it tackles the problem of complexity. It decomposes what would otherwise be a monstrous monolithic application into a set of services. While the total amount of functionality is unchanged, the application has been broken up into manageable chunks or services. Each service has a well-defined boundary in the form of a remote procedure call (RPC)-driven or message-driven API. The Microservices Architecture pattern enforces a level of modularity that in practice is extremely difficult to achieve with a monolithic code base. Consequently, individual services are much faster to develop, and much easier to understand and maintain.

b. Second, this architecture enables each service to be developed independently by a team that is focused on that service. The developers are free to choose whatever technologies make sense, provided that the service honors the API contract. Of course, most organizations would want to avoid complete anarchy by limiting technology options. However, this freedom means that developers are no longer obligated to use the possibly obsolete technologies that existed at the start of a new project. When writing a new service, they have the option of using current technology. Moreover, since services are relatively small, it becomes more feasible to rewrite an old service using current technology.

c. Third, the Microservices Architecture pattern enables each microservice to be deployed independently. Developers never need to coordinate the deployment of changes that are local to their service. These kinds of changes can be deployed as soon as they have been tested. The UI team can, for example, perform A|B testing and rapidly iterate on UI changes. The Microservices Architecture pattern makes continuous deployment possible.

d. Finally, the Microservices Architecture pattern enables each service to be scaled independently. You can deploy just the number of instances of each service that satisfy its capacity and availability constraints. Moreover, you can use the hardware that best matches a service’s resource requirements. For example, you can deploy a CPU-intensive image processing service on EC2 Compute Optimized instances and deploy an in-memory database service on EC2 Memory-optimized instances.

19. Drawbacks of Microservice design:

a. Service Size: One drawback is the name itself. The term microservice places excessive emphasis on service size. In fact, there are some developers who advocate for building extremely fine-grained 10-100 LOC services. While small services are preferable, it’s important to remember that small services are a means to an end, and not the primary goal. The goal of microservices is to sufficiently decompose the application in order to facilitate agile application development and deployment.

b. Another major drawback of microservices is the complexity that arises from the fact that a microservices application is a distributed system. Developers need to choose and implement an inter-process communication mechanism based on either messaging or RPC. Moreover, they must also write code to handle partial failure, since the destination of a request might be slow or unavailable. While none of this is rocket science, it’s much more complex than in a monolithic application, where modules invoke one another via language-level method/procedure calls.

c. Another challenge with microservices is the partitioned database architecture. Business transactions that update multiple business entities are fairly common. These kinds of transactions are trivial to implement in a monolithic application because there is a single database. In a microservices-based application, however, you need to update multiple databases owned by different services. Using distributed transactions is usually not an option, and not only because of the CAP theorem. They simply are not supported by many of today’s highly scalable NoSQL databases and messaging brokers. You end up having to use an eventual consistency-based approach, which is more challenging for developers.

d. Testing a microservices application is also much more complex. For example, with a modern framework such as Spring Boot, it is trivial to write a test class that starts up a monolithic web application and tests its REST API. In contrast, a similar test class for a service would need to launch that service and any services that it depends upon, or at least configure stubs for those services. Once again, this is not rocket science, but it’s important to not underestimate the complexity of doing this.

e. Multi-service Change: Another major challenge with the Microservices Architecture pattern is implementing changes that span multiple services. For example, let’s imagine that you are implementing a story that requires changes to services A, B, and C, where A depends upon B and B depends upon C. In a monolithic application you could simply change the corresponding modules, integrate the changes, and deploy them in one go. In contrast, in a Microservices Architecture pattern you need to carefully plan and coordinate the rollout of changes to each of the services. For example, you would need to update service C, followed by service B, and then finally service A. Fortunately, most changes typically impact only one service; multi-service changes that require coordination are relatively rare.

f. Deployment: Deploying a microservices-based application is also much more complex. A monolithic application is simply deployed on a set of identical servers behind a traditional load balancer. Each application instance is configured with the locations (host and ports) of infrastructure services such as the database and a message broker. In contrast, a microservice application typically consists of a large number of services. For example, Hailo has 160 different services and Netflix has more than 600, according to Adrian Cockcroft.

g. Each service will have multiple runtime instances. That’s many more moving parts that need to be configured, deployed, scaled, and monitored. In addition, you will also need to implement a service discovery mechanism that enables a service to discover the locations (hosts and ports) of any other services it needs to communicate with.

20. Scale Cube:

X-axis: Scale by Cloning -> Horizontal duplication

Y-axis: Scale by splitting different things ->Functional decomposition (Microservice design)

Z-axis: Scale by splitting similar things -> Data Partitioning

21. A|B Testing:

A/B testing (sometimes called split testing) is comparing two versions of a web page to see which one performs better. You compare two web pages by showing the two variants (let's call them A and B) to similar visitors at the same time. The one that gives a better conversion rate, wins!

22. IPC (Inter-Process Communication):

When selecting an IPC mechanism for a service, it is useful to think first about how services interact. There are a variety of client <-> service interaction styles. They can be categorized along two dimensions. The first dimension is whether the interaction is one-to-one or one-to-many:

• One-to-one – Each client request is processed by exactly one service instance.

• One-to-many – Each request is processed by multiple service instances.

The second dimension is whether the interaction is synchronous or asynchronous:

• Synchronous – The client expects a timely response from the service and might even block while it waits.

• Asynchronous – The client doesn’t block while waiting for a response, and the response, if any, isn’t necessarily sent immediately.

23. Event-Driven Architecture:

In a microservices architecture, each microservice has its own private datastore. Different microservices might use different SQL and NoSQL databases. While this database architecture has significant benefits, it creates some distributed data management challenges. The first challenge is how to implement business transactions that maintain consistency across multiple services. The second challenge is how to implement queries that retrieve data from multiple services.

For many applications, the solution is to use an event-driven architecture. One challenge with implementing an event-driven architecture is how to atomically update state and how to publish events. There are a few ways to accomplish this, including using the database as a message queue, transaction log mining, and event sourcing.

In this architecture, a microservice publishes an event when something notable happens, such as when it updates a business entity. Other microservices subscribe to those events. When a microservice receives an event it can update its own business entities, which might lead to more events being published.

Benefits:

- It enables the implementation of transactions that span multiple services and provide eventual consistency.

- Another benefit is that it also enables an application to maintain materialized views.

Drawbacks:

- One drawback is that the programming model is more complex than when using ACID transactions. Often you must implement compensating transactions to recover from application-level failures; for example, you must cancel an order if the credit check fails. Also, applications must deal with inconsistent data. That is because changes made by in-flight transactions are visible.

- The application can also see inconsistencies if it reads from a materialized view that is not yet updated.

- Another drawback is that subscribers must detect and ignore duplicate events.

24. Optimistic vs Pessimistic Locking:

Optimistic: Do not acquire any locks. When you are ready to commit a transaction, at that point you see if no other transaction updated the record you are working on.

Pessimistic: Acquire the locks before hand and then you commit your transaction.

25. Caching: Used to speed up your request. If you know that some data is accessed more frequently then cache it.

Caches in a system can be implemented in two ways:

- Every node has its own cache.

- Distributed Cache: One cache for all.

Things to consider for Cache:

- Cache can not the source of truth

- Cache data has to be pretty small since everything is stored in memory.

Caches are of two types:

- Write through cache: System writes into the cache and then into the database so that cache contains all the recent data.

- Write back cache: System writes into the database and then into the cache.

26. Data Centers: Data Centers has Racks and Racks has hosts.

27. HTTP vs HTTP2 vs WebSockets:

- HTTP: Request -Response type of architecture. Entire web runs on it.

- HTTP2: This is better http and fixes some things which are bad in http. It can do multiple requests over a single connection.

- Web Sockets: Fully bi-directional communication between client and server.

28. Synchronous/Blocking: When you call a function or API and that blocks the thread which you’re calling until it executes totally, then it is synchronous.

Asynchronous/Non-Blocking: When you call a function or API and that doesn’t block the thread which you’re calling, then it is asynchronous.

**Operating Systems**

1. Process vs Thread:

a. The typical difference is that threads (of the same process) run in a shared memory space, while processes run in separate memory spaces.

The benefits of this is:

\* For tasks that involve sharing large amount of data, the fact that threads all share a process's memory pool, they do not have separate copies, that means that different threads can read and modify a shared pool of memory easily. While in case of processes data is shared through inter process communication (e.g., message passing mechanism) which significantly increases the number of system calls.

The disadvantages of this is:

\* Synchronization overhead in case of thread: Shared data that is modified requires special handling in the form of locks and mutexes to prevent race conditions.

\* If something goes wrong in one thread and causes data corruption, the data is corrupted for all the threads in that process.

b. Time required for creation and termination of process is more than a thread: To start a process, the whole process area must be duplicated for a new process copy to start, while in case of thread, very little memory coping is required(just the thread stack)

c. Resource consumption is more in a process compared to a thread: A process operations are controlled with the help of PCB(Process control Block) can be considered as the brain of the process, which contains all the crucial information regarding to a process such as a process id, priority, state(ready, running, blocked and terminated), PWS and contents CPU register. For Threads, the kernel allocates a stack and a thread control block (TCB) which contains registers.

d. Context switching in a process is faster in a thread than a process: The CPU caches and program context can be maintained between threads in a process while in case of processes it needs to be reloaded for switching a CPU to different processes.

e. When a process is terminated uncertainly, it results in loss of process, while when a thread is terminated uncertainly, it can be reclaimed.

2. Virtual Memory:

Virtual memory is a memory management capability of an operating system (OS) that uses hardware and software to allow a computer to compensate for physical memory shortages by temporarily transferring data from random access memory (RAM) to disk storage.

Virtual memory was developed at a time when physical memory -- the installed RAM -- was expensive. Computers have a finite amount of RAM, so memory can run out, especially when multiple programs run at the same time. A system using virtual memory uses a section of the hard drive to emulate RAM. With virtual memory, a system can load larger programs or multiple programs running at the same time, allowing each one to operate as if it has infinite memory and without having to purchase more RAM.

The use of virtual memory has its tradeoffs, particularly with speed. It's generally better to have as much physical memory as possible so programs work directly from RAM or physical memory. The use of virtual memory slows a computer because data must be mapped between virtual and physical memory, which requires extra hardware support for address translations.

3. Difference between virtual memory and swap memory:

Virtual memory is a combination of RAM and disk space that running processes can use.

Swap space is the portion of virtual memory that is on the hard disk, used when RAM is full.

4. IPv4 vs IPv6: IPv4 has 32 bit ddresses while IPv6 has 128bit addresses.

5. Concurrency vs Parallelism:

Parallelism

Using multiple computational resources (like more processor cores) to perform a computation faster, usually executing at the same time.

Example: summing a list of Integers by dividing it in half and calculating both halves in parallel.

Main concern: efficiency.

Concurrency

Multiple tasks interleaved. Concurrency doesn't have to be multithreaded. We can write concurrent applications on single processor using methods such as event loops.

Example: Communicating with external services through HTTP.

Main concern: interaction with multiple, independent and external agents.

CPU-bound task

Operation that mostly requires processor resources to finish its computation.

IO-bound task

Operation that mostly does I/O and it doesn't depend on your computation resources, e.g. waiting for disk operation to finish or external service to answer your request.

**Databases**

1. ACID Properties:

A transaction is a single logical unit of work which accesses and possibly modifies the contents of a database. Transactions access data using read and write operations.

In order to maintain consistency in a database, before and after transaction, certain properties are followed. These are called ACID properties.

Atomicity

By this, we mean that either the entire transaction takes place at once or doesn’t happen at all. There is no midway i.e. transactions do not occur partially. Each transaction is considered as one unit and either runs to completion or is not executed at all. It involves following two operations.

—Abort: If a transaction aborts, changes made to database are not visible.

—Commit: If a transaction commits, changes made are visible.

Atomicity is also known as the ‘All or nothing rule’.

Consistency

This means that integrity constraints must be maintained so that the database is consistent before and after the transaction. It refers to correctness of a database.

Isolation

This property ensures that multiple transactions can occur concurrently without leading to inconsistency of database state. Transactions occur independently without interference. Changes occurring in a particular transaction will not be visible to any other transaction until that particular change in that transaction is written to memory or has been committed. This property ensures that the execution of transactions concurrently will result in a state that is equivalent to a state achieved these were executed serially in some order.

This actually means isolating some users from reads/write for small amt of time so that the database stays consistent.

Durability

This property ensures that once the transaction has completed execution, the updates and modifications to the database are stored in and written to disk and they persist even if system failure occurs. These updates now become permanent and are stored in a non-volatile memory. The effects of the transaction, thus, are never lost.

The ACID properties, in totality, provide a mechanism to ensure correctness and consistency of a database in a way such that each transaction is a group of operations that acts a single unit, produces consistent results, acts in isolation from other operations and updates that it makes are durably stored.

2. 2-Phase Commit / 2PC:

Database vendors long ago recognized the need for partitioning databases and introduced a technique known as 2PC (two-phase commit) for providing ACID guarantees across multiple database instances. The protocol is broken into two phases:

\* First, the transaction coordinator asks each database involved to precommit the operation and indicate whether commit is possible. If all databases agree the commit can proceed, then phase 2 begins.

\* The transaction coordinator asks each database to commit the data.

If any database vetoes the commit, then all databases are asked to roll back their portions of the transaction. What is the shortcoming? We are getting consistency across partitions and partitioning as well. So, we are obviously sacrificing availability (CAP theorem)

3. BASE (basically available, soft state, eventually consistent): An ACID Alternative

In partitioned databases, trading some consistency for availability can lead to dramatic improvements in scalability.

If ACID provides the consistency choice for partitioned databases, then how do you achieve availability instead? One answer is BASE (basically available, soft state, eventually consistent).

BASE is diametrically opposed to ACID. Where ACID is pessimistic and forces consistency at the end of every operation, BASE is optimistic and accepts that the database consistency will be in a state of flux. Although this sounds impossible to cope with, in reality it is quite manageable and leads to levels of scalability that cannot be obtained with ACID.

The availability of BASE is achieved through supporting partial failures without total system failure. Here is a simple example: if users are partitioned across five database servers, BASE design encourages crafting operations in such a way that a user database failure impacts only the 20 percent of the users on that particular host. There is no magic involved, but this does lead to higher perceived availability of the system.

4. Double Booking Problem:

https://www.youtube.com/watch?v=\_95dCYv2Xv4&ab\_channel=HusseinNasser

https://www.youtube.com/watch?v=gv62vmvyy6s&list=UUMO\_ML5xP23TOWKUcc-oAE\_Eg&index=49&ab\_channel=HusseinNasser

In applications like Book my show, flight reservation system, etc when we want to book seats, what we generally do is

a. Check if the selected seat is available

SELECT \* from seats where seatId = .. AND isBooked = 0

b. If (a) returns true, then we update the seat with the user details and basically book it for him/her

UPDATE seats set isBooked=1, name = ... WHERE seatId = ..

If (b) returns false, then we respond to the user saying "Seat Already Booked"

But this code is not correct and has a problem of 'Race Condition' which will cause a Double Booking Problem. Here's how:

Imagine that the application is having huge traffic and multiple people are trying to book the tickets. It may happen that two people are trying to book the same seat at the same time. When this happens, both user A and B check for Step (a) together and then get that the seat is available for them to book. Both the processes will now continue and update the seat as Booked. The one which updates it later is actually reserving the seats actually but both of them would get the response saying "Tickets booked successfully" with the seat numbers. Now this becomes a problem since the same seat is given to both the users A and B.

How to prevent Double Booking?

The first process to get to the row should obtain an exclusive lock on that row so that the other process when it tries to execute it will have to wait for accessing this row until the other trasaction actually completes. We do it like this:

SELECT \* from seats where seatId = .. AND isBooked = 0 FOR UPDATE

Using the statement "FOR UPDATE" we are telling the database to obtain a row level lock so that if other process comes, it will make them wait. We can also put a timeout here which will make the other process wait for X secs and then allow it. We are sure that after X seconds my transaction would be completed so when the other process comes to read it, it would say "Seats booked already".

This row level locks are available in some certain databases only like postgres, mysql, etc.

If we do not use the time thing for X secs, the other process is going to wait for ever till the first transaction completes. This is bad since we should never want a process to wait indefinitely. So, it is better to provide a timeout so that the other processes is allowed to enter after X secs. We can do it like this:

SET LOCAL lock\_timeout = '1s';

This is now a bullet proof online seat booking system. It will not have a Double Booking or Race Condition issue. User which reaches first obtains a lock on the specific row and the other User keeps waiting till this transaction 1 is completed. Once the transaction 1 is completed, the lock is released and the user 2 can now access the row which would say isBooked = 1 and the user would get the message that seat is already booked.

5. Exclusive Lock and Shared Lock:

https://www.youtube.com/watch?v=b7razfltSFM&ab\_channel=HusseinNasser

Exclusive Lock: I want to read a piece of data in the database but I want to make sure that when I read/update this value, no body is attempting to read/update this value. I want to acquire exclusive lock on this row. I will be only one who is allowed to update/read the value. While there is an exclusive lock, no body can read or write or be connected.

This is used to ensure consistency.

When you take a exclusive lock on a row, there should be NO shared locks on it.

Shared Locks: I want to read a value from the database I want to make sure that no body change that. If any body is trying to update that value, they will fail. More people can read but no one can update.

When you take a shared lock on a row, there should be NO exclusive locks on it.

6. Pessimistic concurrency control vs Optimistic concurrency control

https://www.youtube.com/watch?v=I8IlO0hCSgY&ab\_channel=HusseinNasser

https://www.youtube.com/watch?v=H\_zJ81I\_D5E&ab\_channel=HusseinNasser

Pessimistic concurrency control: When a transaction is happening for updating a value, an exclusive lock is taken for that row so that no other process can read/update the data in the same row. This is in order to get data consistent at all point of time.

Cons:

a. When we have millions of transactions updating the same row (Bad Database design though! Ehh!) then everything will be slow.

b. Locks are expensive. Locks are difficult to handle.

Optimistic concurrency control: When a transaction is happening for updating a value, no locks are taken at all. Let both the transaction book the same seat. Before commiting the transaction, it just says Check if the data has changed already, if it has changed already, then it means that someone else has already changed the value at this row, so rollback whatever part of the transaction was done and start over. If the data has not changed, then go ahead and commit the transaction.

Cons:

a. It is still risky since there may be a change that it done after the "revised read" and commiting the transaction so still there exists a time window (time-to-check to time-to-read: https://en.wikipedia.org/wiki/Time-of-check\_to\_time-of-use) which will allow some race conditions to happen and thus our data base would be inconsistent. This is actually not even a lock, it is something that has to be handled at application level.

Dynamo db uses optimistic locking concept. It adds an extra field called version number to each object and the version number gets incremented with each update to that item. If someone tries to update the item with some stale version a recoverable exception is thrown so there should be retries at application level.

Problem with this approach in case of multiple threads updating the same item, a particular thread can receive version conflict exception again and again, because of limited number of retries at application layer that thread can fail altogether.

Ref: https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/DynamoDBMapper.OptimisticLocking.html

PostGres - Puts these locks on disks

SQL Server - Handles these locks in memory

MongoDB - has row/document level locks for writes and automatic retrying.

**Object Oriented Programming / C++**

1. Inheritance:

The capability of a class to derive properties and characteristics from another class is called Inheritance. Inheritance is one of the most important feature of Object Oriented Programming.

Sub Class: The class that inherits properties from another class is called Sub class or Derived Class.

Super Class:The class whose properties are inherited by sub class is called Base Class or Super class.

class subclass\_name : access\_mode base\_class\_name

{

//body of subclass

};

Here, subclass\_name is the name of the sub class, access\_mode is the mode in which you want to inherit this sub class for example: public, private etc. and base\_class\_name is the name of the base class from which you want to inherit the sub class.

Note: A derived class doesn’t inherit access to private data members. However, it does inherit a full parent object, which contains any private members which that class declares

Modes of Inheritance

- Public mode: If we derive a sub class from a public base class. Then the public member of the base class will become public in the derived class and protected members of the base class will become protected in derived class.

- Protected mode: If we derive a sub class from a Protected base class. Then both public member and protected members of the base class will become protected in derived class.

- Private mode: If we derive a sub class from a Private base class. Then both public member and protected members of the base class will become Private in derived class.

Multiple Inheritance

Multiple Inheritance is a feature of C++ where a class can inherit from more than one classes. i.e one sub class is inherited from more than one base classes.

Syntax:

class subclass\_name : access\_mode base\_class1, access\_mode base\_class2, ....

{

//body of subclass

};

2. Polymorphism:

The word polymorphism means having many forms. In simple words, we can define polymorphism as the ability of a message to be displayed in more than one form.

Real life example of polymorphism, a person at a same time can have different characteristic. Like a man at a same time is a father, a husband, a employee. So a same person posses have different behavior in different situations. This is called polymorphism.

In C++ polymorphism is mainly divided into two types:

\* Compile time Polymorphism:

This type of polymorphism is achieved by function overloading or operator overloading.

- Function Overloading:

When there are multiple functions with same name but different parameters then these functions are said to be overloaded. Functions can be overloaded by change in number of arguments or/and change in type of arguments.

- Operator Overloading:

C++ also provide option to overload operators. For example, we can make the operator (‘+’) for string class to concatenate two strings. We know that this is the addition operator whose task is to add to operands. So a single operator ‘+’ when placed between integer operands , adds them and when placed between string operands, concatenates them.

\* Runtime Polymorphism / Virtual Functions:

https://www.geeksforgeeks.org/virtual-functions-and-runtime-polymorphism-in-c-set-1-introduction/

This type of polymorphism is achieved by Function Overriding.

Function overriding on the other hand occurs when a derived class has a definition for one of the member functions of the base class. That base function is said to be overridden.

The idea is, virtual functions are called according to the type of object pointed or referred, not according to the type of pointer or reference. In other words, virtual functions are resolved late, at runtime.

class Base

{

public:

virtual void show() { cout<<" In Base \n"; }

};

class Derived: public Base

{

public:

void show() { cout<<"In Derived \n"; }

};

int main(void)

{

Base \*bp = new Derived;

bp->show(); // RUN-TIME POLYMORPHISM

return 0;

}

This will call show() in derived class and not base class. So, this depends on the object pointed and not on the type of object pointer. This is the reason it is not resolved while compilation but in runtime since anything can be present in place of derived.

Virtual functions are so useful that later languages like Java keep all methods as virtual by default.

How does Polymorphism work at runtime?

MUST WATCH: https://www.youtube.com/watch?v=Z\_FiER8aAqM

Compiler maintains two things to this:

vtable: A table of function pointers. It is maintained per class.

(Base -> show(),.. Derived->show(),..)

vptr: A pointer to vtable. It is maintained per object.

vptr is made for all objects like bp which are of class Derived.

Compiler adds additional code at two places to maintain and use vptr.

- Code in every constructor. This code sets vptr of the object being created. This code sets vptr to point to vtable of the class.

- Code with polymorphic function call (e.g. bp->show() in above code). Wherever a polymorphic call is made, compiler inserts code to first look for vptr using base class pointer or reference (In the above example, since pointed or referred object is of derived type, vptr of derived class is accessed). Once vptr is fetched, vtable of derived class can be accessed. Using vtable, address of derived derived class function show() is accessed and called.

3. Pure Virtual Function:

A pure virtual function (or abstract function) in C++ is a virtual function for which we don’t have implementation, we only declare it. A pure virtual function is declared by assigning 0 in declaration.

// An abstract class

class Test

{

// Data members of class

public:

// Pure Virtual Function

virtual void show() = 0;

};

**Similarities/Disimilarities between Pure Virtual Functions and Virtual Functions**

Virtual Function in C++

A virtual function is a member function which is declared within a base class and is re-defined(Overriden) by a derived class. When you refer to a derived class object using a pointer or a reference to the base class, you can call a virtual function for that object and execute the derived class’s version of the function.

Pure Virtual Functions in C++

A pure virtual function (or abstract function) in C++ is a virtual function for which we don’t have an implementation, we only declare it. A pure virtual function is declared by assigning 0 in the declaration.

Similarities between virtual function and pure virtual function

These are the concepts of Run-time polymorphism.

Prototype i.e. Declaration of both the functions remains the same throughout the program.

These functions can’t be global or static.

4. Abstract Classes: Sometimes implementation of all function cannot be provided in a base class because we don’t know the implementation. Such a class is called abstract class. For example, let Shape be a base class. We cannot provide implementation of function draw() in Shape, but we know every derived class must have implementation of draw(). Similarly an Animal class doesn’t have implementation of move() (assuming that all animals move), but all animals must know how to move. We cannot create objects of abstract classes.

- A class is abstract if it has at least one pure virtual function.

- We can have pointers and references of abstract class type.

- If we do not override the pure virtual function in derived class, then derived class also becomes abstract class.

5. Encapsulation:

Encapsulation is defined as wrapping up of data and information under a single unit. In Object Oriented Programming, Encapsulation is defined as binding together the data and the functions that manipulates them.

There are 3 types of access modifiers available in C++:

- Public: All the class members declared under public will be available to everyone. The data members and member functions declared public can be accessed by other classes too. The public members of a class can be accessed from anywhere in the program using the direct member access operator (.) with the object of that class.

- Private: The class members declared as private can be accessed only by the functions inside the class. They are not allowed to be accessed directly by any object or function outside the class. Only the member functions or the friend functions are allowed to access the private data members of a class.

- Protected: Protected access modifier is similar to that of private access modifiers, the difference is that the class member declared as Protected are inaccessible outside the class but they can be accessed by any subclass(derived class) of that class.

Note: If we do not specify any access modifiers for the members inside the class then by default the access modifier for the members will be Private.

6. Data Abstraction:

Data abstraction is one of the most essential and important feature of object oriented programming in C++. Abstraction means displaying only essential information and hiding the details. Data abstraction refers to providing only essential information about the data to the outside world, hiding the background details or implementation.

Consider a real life example of a man driving a car. The man only knows that pressing the accelerators will increase the speed of car or applying brakes will stop the car but he does not know about how on pressing accelerator the speed is actually increasing, he does not know about the inner mechanism of the car or the implementation of accelerator, brakes etc in the car. This is what abstraction is.

In C++, it can be implemented using:

- Abstraction using Classes

- Abstraction in Header files

- Abstraction using access specifiers: public, private and protected

7. Friend Class / Friend Function:

A friend class can access private and protected members of other class in which it is declared as friend. It is sometimes useful to allow a particular class to access private members of other class.

Friend Function Like friend class, a friend function can be given special grant to access private and protected members.

1. What is the difference between struct and class?

The data members/attributes of a struct by default are public while the data members/attributes of a class are private by default. It is a good practice to use struct when you have only data members. When you have function members as well, then it is better to use classes.

2. What is the best practice to declare, define and use functions in C++?

Declarations, Definitions and using functions all together in one file is not considered a good practice.

It is best practice to write down the function declarations within the classes in one file, Then function definitions in another file. Then, use those functions in another file.

3. What is difference between struct and union?

'struct' reserves memory all its data types/members defined underneath it. 'union' reserves memory for only the maximum data type/members defined in it.

'struct' is used when we want to access all the members/attributes in it. 'union' is used when we want to access only one out of many data type/members defined in it. E.g., You would define gender as a union since only one out of male, female would be true and we would need only one of them to be set.So, reserving space for any one of them would make sense.

**General Programming / C**

1. Memory Layout of a C program: https://www.geeksforgeeks.org/memory-layout-of-c-program/

A typical memory representation of C program consists of following sections.

a. Text segment: A text segment , also known as a code segment or simply as text, is one of the sections of a program in an object file or in memory, which contains executable instructions.

b. Initialized data segment: Initialized data segment, usually called simply the Data Segment. A data segment is a portion of virtual address space of a program, which contains the global variables and static variables that are initialized by the programmer. Ex: static int i = 10 will be stored in data segment and global int i = 10 will also be stored in data segment

c. Uninitialized data segment: Uninitialized data segment, often called the “bss” segment, named after an ancient assembler operator that stood for “block started by symbol.” Data in this segment is initialized by the kernel to arithmetic 0 before the program starts executing. For instance a variable declared static int i; would be contained in the BSS segment.

d. Stack: Stack, where automatic variables are stored, along with information that is saved each time a function is called. Each time a function is called, the address of where to return to and certain information about the caller’s environment, such as some of the machine registers, are saved on the stack. The newly called function then allocates room on the stack for its automatic and temporary variables. This is how recursive functions in C can work. Each time a recursive function calls itself, a new stack frame is used, so one set of variables doesn’t interfere with the variables from another instance of the function.

e. Heap: Heap is the segment where dynamic memory allocation usually takes place. The heap area begins at the end of the BSS segment and grows to larger addresses from there.The Heap area is managed by malloc, realloc, and free, which may use the brk and sbrk system calls to adjust its size

2. Direct Address Table:

Direct Address Table is a data structure that has the capability of mapping records to their corresponding keys using arrays. In direct address tables, records are placed using their key values directly as indexes. They facilitate fast searching, insertion and deletion operations.

We can understand the concept using the following example. We create an array of size equal to maximum value plus one (assuming 0 based index) and then use values as indexes. For example, in the following diagram key 21 is used directly as index.

Advantages:

- Searching in O(1) Time: Direct address tables use arrays which are random access data structure, so, the key values (which are also the index of the array) can be easily used to search the records in O(1) time.

- Insertion in O(1) Time: We can easily insert an element in an array in O(1) time. The same thing follows in a direct address table also.

- Deletion in O(1) Time: Deletion of an element takes O(1) time in an array. Similarly, to delete an element in a direct address table we need O(1) time.

Disadvantages:

- Prior knowledge of maximum key value

- Practically useful only if the maximum value is very less.

- It causes wastage of memory space if there is a significant difference between total records and maximum value.

- Hashing can overcome these limitations of direct address tables.

Difference between Direct Address Table and Hashing:

Hashing is an improvement over Direct Access Table. The only difference from hashing is, we do not use a hash function to find the index. We rather directly use values as indexes.

3. Hashing:

Hash Function: A function that converts a given big phone number to a small practical integer value. The mapped integer value is used as an index in hash table. In simple terms, a hash function maps a big number or string to a small integer that can be used as index in hash table.

A good hash function should have following properties:

1) Efficiently computable.

2) Should uniformly distribute the keys (Each table position equally likely for each key)

Hash Table: An array that stores pointers to records corresponding to a given phone number.

Collision Handling: Since a hash function gets us a small number for a big key, there is possibility that two keys result in same value. The situation where a newly inserted key maps to an already occupied slot in hash table is called collision and must be handled using some collision handling technique. Following are the ways to handle collisions:

- Separate Chaining: The idea is to make each cell of hash table point to a linked list of records that have same hash function value. Chaining is simple, but requires additional memory outside the table.

Advantages:

1) Simple to implement.

2) Hash table never fills up, we can always add more elements to chain.

3) Less sensitive to the hash function or load factors.

4) It is mostly used when it is unknown how many and how frequently keys may be inserted or deleted.

Disadvantages:

1) Cache performance of chaining is not good as keys are stored using linked list. Open addressing provides better cache performance as everything is stored in same table.

2) Wastage of Space (Some Parts of hash table are never used)

3) If the chain becomes long, then search time can become O(n) in worst case.

4) Uses extra space for links.

- Open Addressing: In open addressing, all elements are stored in the hash table itself. Each table entry contains either a record or NIL. When searching for an element, we one by one examine table slots until the desired element is found or it is clear that the element is not in the table.

Read https://www.geeksforgeeks.org/hashing-set-3-open-addressing/ for more details.

Double hashing is a collision resolving technique in Open Addressed Hash tables. Double hashing uses the idea of applying a second hash function to key when a collision occurs.

Applications of hashing here: https://www.geeksforgeeks.org/applications-of-hashing/

4. If we are using separate chaining and we have a linked list of records mapped to a cell in the hash table, then how do I figure out what is the value of the key I asked for? Ex: cat and dog both give 123 as the output from the has function. { cat=>1 , dog=> 2}. If I search for cat, how will I get 1?

> In linked list we don't just save the value, but also save the key. So, when we look for "cat" and the hashing function gives me "123", then in the linked list, I will search for a value whose key is "cat". Hashing helps here since the search space reduced to just the linked list attached with the address "123"

5. Compiling a C program:

There are four phases for a C program to become an executable:

\* Pre-processing: This is the first phase through which source code is passed. This phase include:

+ Removal of Comments

+ Expansion of Macros

+ Expansion of the included files.

The preprocessed output is stored in the filename.i.

\* Compilation: The next step is to compile filename.i and produce an; intermediate compiled output file filename.s. This file is in assembly level instructions.

\* Assembly: In this phase the filename.s is taken as input and turned into filename.o by assembler. This file contain machine level instructions. At this phase, only existing code is converted into machine language, the function calls like printf() are not resolved.

\* Linking: This is the final phase in which all the linking of function calls with their definitions are done. Linker knows where all these functions are implemented. Linker does some extra work also, it adds some extra code to our program which is required when the program starts and ends. For example, there is a code which is required for setting up the environment like passing command line arguments.

A linker can accomplish this task in two ways, by copying the code of library function to your object code, or by making some arrangements so that the complete code of library functions is not copied, but made available at run-time.

+ Static Linking: Static Linking and Static Libraries is the result of the linker making copy of all used library functions to the executable file. Static Linking creates larger binary files, and need more space on disk and main memory. Examples of static libraries (libraries which are statically linked) are, .a files in Linux and .lib files in Windows.

- For a static library, the actual code is extracted from the library by the linker and used to build the final executable at the point you compile/build your application.

- One major advantage of static libraries being preferred even now “is speed”. There will be no dynamic querying of symbols in static libraries. Many production line software use static libraries even today.

- One drawback of static libraries is, for any change(up-gradation) in the static libraries, you have to recompile the main program every time.

- Each process gets its own copy of the code and data. Where as in case of dynamic libraries it is only code shared, data is specific to each process.

+ Dynamic Linking: Dynamic linking and Dynamic Libraries Dynamic Linking doesn’t require the code to be copied, it is done by just placing name of the library in the binary file. The actual linking happens when the program is run, when both the binary file and the library are in memory. Examples of Dynamic libraries (libraries which are linked at run-time) are, .so in Linux and .dll in Windows.

Q. Whether stdio.h is linked statically or dynamically?

stdio.h is not library, it just a header file which contains the declaration. For example printf's only declaration is present in stdio.h but its definition is in the libc.so.(shared object file) so by default it is dynamically linked.

If you want it to be static, then do ..$ gcc -static filename.c

Q. Difference between header files and library files?

Header contains the function signature, but library stores much more information about a function. It stores function definition also. Also the path of the linked other libraries are stored in libraries.

6. Dangling Pointers / Void Pointers / NULL Pointers / Wild Pointers:

a. Dangling Pointers: A pointer pointing to a memory location that has been deleted (or freed) is called dangling pointer. There are three different ways where a pointer acts as dangling pointer:

\* De-allocation of memory:

int \*ptr = (int \*)malloc(sizeof(int));

// After below free call, ptr becomes a dangling pointer

free(ptr);

// No more a dangling pointer

ptr = NULL;

\* Function Call: If a function returns the address of a local variable, then that address would garbage value and the pointer in the main function would be a dangling pointer.

\* Variable goes out of scope: In another scope, the variable is reachable and is assigned something which isn't visible in the outer scope.

int \*ptr;

.....

{

int ch;

ptr = &ch;

}

.....

// Here ptr is dangling pointer

b. Void Pointers: Void pointer is a specific pointer type – void \* – a pointer that points to some data location in storage, which doesn’t have any specific type. Void refers to the type. Basically the type of data that it points to is can be any. If we assign address of char data type to void pointer it will become char Pointer, if int data type then int pointer and so on. Any pointer type is convertible to a void pointer hence it can point to any value.

c. NULL Pointers: NULL Pointer is a pointer which is pointing to nothing. In case, if we don’t have address to be assigned to a pointer, then we can simply use NULL.

// Null Pointer

int \*ptr = NULL;

printf("The value of ptr is %u", ptr);

- NULL vs Uninitialized pointer – An uninitialized pointer stores an undefined value. A null pointer stores a defined value, but one that is defined by the environment to not be a valid address for any member or object.

d. WILD Pointers: A pointer which has not been initialized to anything (not even NULL) is known as wild pointer. The pointer may be initialized to a non-NULL garbage value that may not be a valid address.

7. What is core dump/ Segmentation Fault?

A core dump is a file of a computer’s documented memory of when a program or computer crashed. The file consists of the recorded status of the working memory at an explicit time, usually close to when the system crashed or when the program ended atypically.

Aside from the entire system memory or just part of the program that aborted, a core dump file may include additional information.

Core Dump/Segmentation fault is a specific kind of error caused by accessing memory that “does not belong to you.”

- When a piece of code tries to do read and write operation in a read only location in memory or freed block of memory, it is known as core dump.

- It is an error indicating memory corruption.

8. Static Variables vs Global Variables:

static int i; // static variable

int j; // global variable

int main(){..}

A global variable has external linkage by default. Its scope can be extended to files other than containing it by giving a matching extern declaration in the other file.

The scope of a global variable can be restricted to the file containing its declaration by prefixing the declaration with the keyword static. Such variables are said to have internal linkage.

NOTE: The keyword static plays a double role. When used in the definitions of global variables, it specifies internal linkage. When used in the definitions of the local variables, it specifies that the lifetime of the variable is going to be the duration of the program instead of being the duration of the function.

9. extern variables / extern functions in C:

Extern functions:

By default, the declaration and definition of a C function have “extern” prepended with them. It means even though we don’t use extern with the declaration/definition of C functions, it is present there.

extern extends the visibility to the whole program, the functions can be used (called) anywhere in any of the files of the whole program provided the declaration of the function is known. (By knowing the declaration of the function, C compiler knows that the definition of the function exists and it goes ahead to compile the program).

Extern Variables:

The extern extends the visibility of the variable to the whole program. By using the extern keyword with a variable we can use the variables anywhere in the program provided we know the declaration of them and the variable is defined somewhere in some other file as a global variable.

There's another important thing associated with extern variables: When extern is used with a variable, it’s only declared not defined.

How would you declare a C variable without defining it?

extern int x;

Here, an integer type variable called var has been declared (remember no definition i.e. no memory allocation for var so far). And we can do this declaration as many times as needed. (remember that declaration can be done any number of times)

Now how would you define a variable?

int var;

Here, an integer type variable called var has been declared as well as defined. (remember that definition is the superset of declaration). Here the memory for var is also allocated.

- As an exception, when an extern variable is declared with initialization, it is taken as the definition of the variable as well.

extern int var = 0;