create a architecture for ola types software:

Creating an architecture for Ola types software involves designing a system that can handle various aspects of a ride-hailing service, such as user management, ride booking, driver management, payment processing, and more. Below is a high-level architecture that outlines how to structure such a system. This architecture assumes the need for scalability, reliability, and maintainability.

**High-Level Architecture for Ola Types Software**

**1. Client Layer**

* **Mobile Apps**: Native apps for iOS and Android for both drivers and riders.
* **Web App**: A web-based interface for users, admin, and support staff.
* **APIs**: Endpoints for third-party integrations and internal services.

**2. Front-End Layer**

* **User Interface (UI)**: For mobile and web applications, handling user interactions.
* **State Management**: Manages the state of the application, handles user sessions, etc.
* **Routing**: Handles navigation within the app.

**3. Backend Layer**

* **Application Server**: Hosts the core application logic.
  + **Microservices Architecture**: Breaks down the application into services like user management, ride management, payment processing, etc.
* **RESTful APIs / GraphQL**: Provides endpoints for mobile and web apps.
* **Authentication & Authorization**: Manages user authentication (e.g., OAuth 2.0) and authorization.

**4. Service Layer**

* **User Management Service**: Handles user profiles, authentication, and authorization.
* **Ride Management Service**: Manages ride requests, driver assignments, ride status updates, etc.
* **Driver Management Service**: Handles driver profiles, availability, and ratings.
* **Payment Service**: Processes payments, manages transactions, and integrates with payment gateways.
* **Notification Service**: Sends out notifications via SMS, email, or push notifications.
* **Analytics Service**: Tracks and analyzes user behavior, ride statistics, and other metrics.

**5. Data Layer**

* **Database**:
  + **Relational Database**: For structured data such as user profiles, ride details, and transactions (e.g., PostgreSQL, MySQL).
  + **NoSQL Database**: For unstructured or semi-structured data like ride history, user preferences (e.g., MongoDB, Cassandra).
* **Cache**: Improves performance and reduces load on the database (e.g., Redis, Memcached).
* **Search Engine**: For efficient searching and filtering of data (e.g., Elasticsearch).

**6. Infrastructure Layer**

* **Load Balancer**: Distributes incoming traffic across multiple servers.
* **API Gateway**: Manages API requests, provides authentication, rate limiting, and request routing.
* **Containerization**: Uses Docker or similar technology to package microservices.
* **Orchestration**: Manages container deployment and scaling (e.g., Kubernetes).
* **Monitoring & Logging**: Tools for tracking system health and debugging issues (e.g., Prometheus, Grafana, ELK Stack).

**7. Security Layer**

* **Encryption**: Ensures data security in transit (e.g., HTTPS) and at rest.
* **Firewalls**: Protects the network and servers from unauthorized access.
* **Intrusion Detection System (IDS)**: Monitors and detects suspicious activities.
* **Vulnerability Management**: Regularly scans for and addresses security vulnerabilities.

**8. DevOps Layer**

* **CI/CD Pipeline**: Automates the build, test, and deployment processes (e.g., Jenkins, GitHub Actions).
* **Version Control**: Manages source code and versioning (e.g., Git).
* **Configuration Management**: Handles configuration settings and environment variables (e.g., Ansible, Terraform).

**9. Integration Layer**

* **Third-Party APIs**: Integration with mapping services (e.g., Google Maps), payment gateways, and other external services.
* **Data Sync**: Ensures data consistency across different services and components.

**Diagram Overview**

1. **Client Layer**
   * Mobile App
   * Web App
   * APIs
2. **Front-End Layer**
   * UI
   * State Management
   * Routing
3. **Backend Layer**
   * Application Server (Microservices)
   * RESTful APIs / GraphQL
   * Authentication & Authorization
4. **Service Layer**
   * User Management Service
   * Ride Management Service
   * Driver Management Service
   * Payment Service
   * Notification Service
   * Analytics Service
5. **Data Layer**
   * Relational Database
   * NoSQL Database
   * Cache
   * Search Engine
6. **Infrastructure Layer**
   * Load Balancer
   * API Gateway
   * Containerization
   * Orchestration
   * Monitoring & Logging
7. **Security Layer**
   * Encryption
   * Firewalls
   * IDS
   * Vulnerability Management
8. **DevOps Layer**
   * CI/CD Pipeline
   * Version Control
   * Configuration Management
9. **Integration Layer**
   * Third-Party APIs
   * Data Sync

Implement a custom dynamic array class that supports basic operations like insertion, deletion, resizing, and clearing.

#include <iostream>

#include <stdexcept>

class DynamicArray {

private:

int\* array;

std::size\_t size;

std::size\_t capacity;

void resize(std::size\_t new\_capacity) {

int\* new\_array = new int[new\_capacity];

for (std::size\_t i = 0; i < size; ++i) {

new\_array[i] = array[i];

}

delete[] array;

array = new\_array;

capacity = new\_capacity;

}

public:

DynamicArray() : array(new int[1]), size(0), capacity(1) {}

~DynamicArray() {

delete[] array;

}

DynamicArray(const DynamicArray& other)

: array(new int[other.capacity]), size(other.size), capacity(other.capacity) {

for (std::size\_t i = 0; i < size; ++i) {

array[i] = other.array[i];

}

}

DynamicArray& operator=(const DynamicArray& other) {

if (this == &other) {

return \*this;

}

delete[] array;

array = new int[other.capacity];

size = other.size;

capacity = other.capacity;

for (std::size\_t i = 0; i < size; ++i) {

array[i] = other.array[i];

}

return \*this;

}

DynamicArray(DynamicArray&& other) noexcept

: array(other.array), size(other.size), capacity(other.capacity) {

other.array = nullptr;

other.size = 0;

other.capacity = 0;

}

DynamicArray& operator=(DynamicArray&& other) noexcept {

if (this == &other) {

return \*this;

}

delete[] array;

array = other.array;

size = other.size;

capacity = other.capacity;

other.array = nullptr;

other.size = 0;

other.capacity = 0;

return \*this;

}

void append(int value) {

if (size == capacity) {

resize(2 \* capacity);

}

array[size++] = value;

}

void insert(std::size\_t index, int value) {

if (index > size) {

throw std::out\_of\_range("Index out of bounds");

}

if (size == capacity) {

resize(2 \* capacity);

}

for (std::size\_t i = size; i > index; --i) {

array[i] = array[i - 1];

}

array[index] = value;

++size;

}

void remove(int value) {

bool found = false;

std::size\_t i = 0;

for (; i < size; ++i) {

if (array[i] == value) {

found = true;

break;

}

}

if (!found) {

throw std::runtime\_error("Value not found in the array");

}

for (std::size\_t j = i; j < size - 1; ++j) {

array[j] = array[j + 1];

}

--size;

if (size > 0 && size == capacity / 4) {

resize(capacity / 2);

}

}

void clear() {

size = 0;

}

int& operator[](std::size\_t index) {

if (index >= size) {

throw std::out\_of\_range("Index out of bounds");

}

return array[index];

}

const int& operator[](std::size\_t index) const {

if (index >= size) {

throw std::out\_of\_range("Index out of bounds");

}

return array[index];

}

std::size\_t getSize() const {

return size;

}

std::size\_t getCapacity() const {

return capacity;

}

void print() const {

for (std::size\_t i = 0; i < size; ++i) {

std::cout << array[i] << " ";

}

std::cout << std::endl;

}

};

int main() {

DynamicArray arr;

arr.append(1);

arr.append(2);

arr.append(3);

std::cout << "Array after appends: ";

arr.print();

arr.insert(1, 10);

std::cout << "Array after inserting 10 at index 1: ";

arr.print();

arr.remove(10);

std::cout << "Array after removing 10: ";

arr.print();

arr.clear();

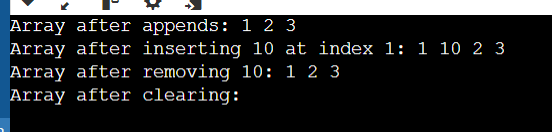
std::cout << "Array after clearing: ";

arr.print();

return 0;

}

Output:



Create a template-based stack class supporting push, pop, and peek operations. Implement it for different data types like int, float, and std::string.

#include <iostream>

#include <stdexcept>

template <typename T>

class Stack {

private:

T\* array;

std::size\_t size;

std::size\_t capacity;

void resize(std::size\_t new\_capacity) {

T\* new\_array = new T[new\_capacity];

for (std::size\_t i = 0; i < size; ++i) {

new\_array[i] = array[i];

}

delete[] array;

array = new\_array;

capacity = new\_capacity;

}

public:

Stack() : array(new T[1]), size(0), capacity(1) {}

~Stack() {

delete[] array;

}

Stack(const Stack& other)

: array(new T[other.capacity]), size(other.size), capacity(other.capacity) {

for (std::size\_t i = 0; i < size; ++i) {

array[i] = other.array[i];

}

}

Stack& operator=(const Stack& other) {

if (this == &other) {

return \*this;

}

delete[] array;

array = new T[other.capacity];

size = other.size;

capacity = other.capacity;

for (std::size\_t i = 0; i < size; ++i) {

array[i] = other.array[i];

}

return \*this;

}

Stack(Stack&& other) noexcept

: array(other.array), size(other.size), capacity(other.capacity) {

other.array = nullptr;

other.size = 0;

other.capacity = 0;

}

Stack& operator=(Stack&& other) noexcept {

if (this == &other) {

return \*this;

}

delete[] array;

array = other.array;

size = other.size;

capacity = other.capacity;

other.array = nullptr;

other.size = 0;

other.capacity = 0;

return \*this;

}

void push(const T& value) {

if (size == capacity) {

resize(2 \* capacity);

}

array[size++] = value;

}

void pop() {

if (size == 0) {

throw std::out\_of\_range("Stack is empty");

}

--size;

if (size > 0 && size == capacity / 4) {

resize(capacity / 2);

}

}

T& peek() {

if (size == 0) {

throw std::out\_of\_range("Stack is empty");

}

return array[size - 1];

}

const T& peek() const {

if (size == 0) {

throw std::out\_of\_range("Stack is empty");

}

return array[size - 1];

}

std::size\_t getSize() const {

return size;

}

bool isEmpty() const {

return size == 0;

}

void print() const {

for (std::size\_t i = 0; i < size; ++i) {

std::cout << array[i] << " ";

}

std::cout << std::endl;

}

};

int main() {

Stack<int> intStack;

intStack.push(1);

intStack.push(2);

intStack.push(3);

std::cout << "Integer Stack: ";

intStack.print();

std::cout << "Peek: " << intStack.peek() << std::endl;

intStack.pop();

std::cout << "After pop: ";

intStack.print();

Stack<float> floatStack;

floatStack.push(1.1f);

floatStack.push(2.2f);

floatStack.push(3.3f);

std::cout << "Float Stack: ";

floatStack.print();

std::cout << "Peek: " << floatStack.peek() << std::endl;

floatStack.pop();

std::cout << "After pop: ";

floatStack.print();

Stack<std::string> stringStack;

stringStack.push("Hello");

stringStack.push("World");

stringStack.push("!");

std::cout << "String Stack: ";

stringStack.print();

std::cout << "Peek: " << stringStack.peek() << std::endl;

stringStack.pop();

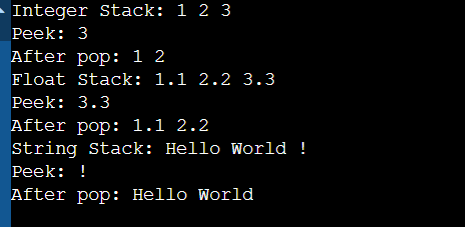
std::cout << "After pop: ";

stringStack.print();

return 0;

}

Output:



Write a program that reads from a file and handles various exceptions such as file not found, read errors, and unexpected data formats.