**Project Report: Event Management System**

**Objective:**

* To create an event management system that provides a basic framework for managing events and equipment associated with those events.
* To Utilize object-oriented programming principles, including inheritance and polymorphism.
* To Implement a user interface with menus and instructions.
* To Ensure that the event management system provides assistance in organizing any event.

**Project Idea Description**:

Event management system is a comprehensive and sophisticated solution designed to streamline and optimize the planning, organization, and execution of events, ranging from small-scale gatherings to large-scale conferences, weddings, corporate meetings, and festivals. This software-based platform leverages technology to simplify the intricate tasks involved in event coordination and offers event planners, organizers, and hosts a centralized hub for managing every aspect of an event's lifecycle. The program uses object-oriented programming (OOP) principles to manage user data and methods efficiently.

**Program Overview:**

This code is an implementation of an Event Management System in C++. It allows users to add, delete, and display various types of events, along with managing equipment for those events. Here's a basic description of the key components and functionalities of the code:Classes: The code defines several classes to model different aspects of the event management system:**Event** is an abstract base class for events. It defines virtual functions for displaying event details, deleting an event, and calculating its cost.

* ***Code Snippet:***

class Event

{

public:

virtual ~Event() {}

virtual void displayDetails() const = 0;

virtual void deleteEvent() = 0;

virtual double calculateCost() const = 0;

};

**EventType** is another abstract base class for event types. It defines a virtual function to display event types.There are concrete classes derived from EventType (e.g., Conference, TradeShow, etc.) that display specific event types.

* ***Code Snippet:***

class EventType

{

public:

virtual ~EventType() {}

virtual void displayEventTypes() const = 0;

};

class Conference : public EventType

{

public:

void displayEventTypes() const override //polymorphism override function

{

cout << "1. Conferences" << endl;

}

};

**EventDetails Class**: Derived from Event, this class represents detailed information about an event, including its name, date, time, location, and the number of participants. It implements the virtual functions of the Event class.

* ***Code Snippet:***

class EventDetails : public Event

{

private:

string name;

string date;

string time;

string location;

int no\_of\_participants;

public:

EventDetails() {} //default constructor called

EventDetails(string n, string d, string t, string l, int participants) //parameterized constructor

: name(n), date(d), time(t), location(l), no\_of\_participants(participants){}

**Equipment Classes**: There are several equipment classes (e.g., Tables, Chairs, etc.) derived from the Equipment base class. These classes represent different types of equipment that can be used for events.

* ***Code Snippet:***

class Equipment

{

public:

virtual ~Equipment() {}

virtual double getCost() const = 0;

};

// Derived classes for different equipment items

class Tables : public Equipment

{

public:

double getCost() const override

{

return 12.0;

}

};

**EquipmentManager Class**: This class is responsible for managing equipment for events. It allows the selection of equipment items and calculates the total cost of selected equipment.

* ***Code Snippet:***

class EquipmentManager

{

private:

map<string, int> selectedEquipment; // Equipment name to quantity

public:

// Function to select equipment

void selectEquipment(const string &equipmentName, int quantity)

{

selectedEquipment[equipmentName] = quantity;

}

double equipmentCost = costPerUnit \* quantity;

totalCost += equipmentCost;

**EventManager Class**: This class manages a collection of events. It allows adding, deleting, and displaying events. It also takes care of freeing memory when an event is deleted.

* ***Code Snippet:***

class EventManager

{

private:

vector<Event \*> events;

public:

~EventManager()

{

for (Event \*event : events)

{

delete event;

}

}

//Function to Add event

void addEvent(Event \*event)

{

events.push\_back(event);

}

//Function to Delete Event

void deleteEvent(int index)

{

if (index >= 0 && index < events.size())

{

delete events[index];

events.erase(events.begin() + index); }

}

void displayEvents() const

{

for (size\_t i = 0; i < events.size(); ++i)

{

events[i]->displayDetails();

cout << "Cost: $" << events[i]->calculateCost() << endl; }

};

Operator Overloading: The code overloads << and >> operators for the EventDetails class to allow input and output of event details.

* ***Code Snippet:***

istream& operator>>(istream& input, EventDetails& event)

{

input.ignore();

getline(input, event.name);

input >> event.date;

input >> event.time;

input.ignore();

getline(input, event.location);

input >> event.no\_of\_participants;

return input;

}

Friend Functions:

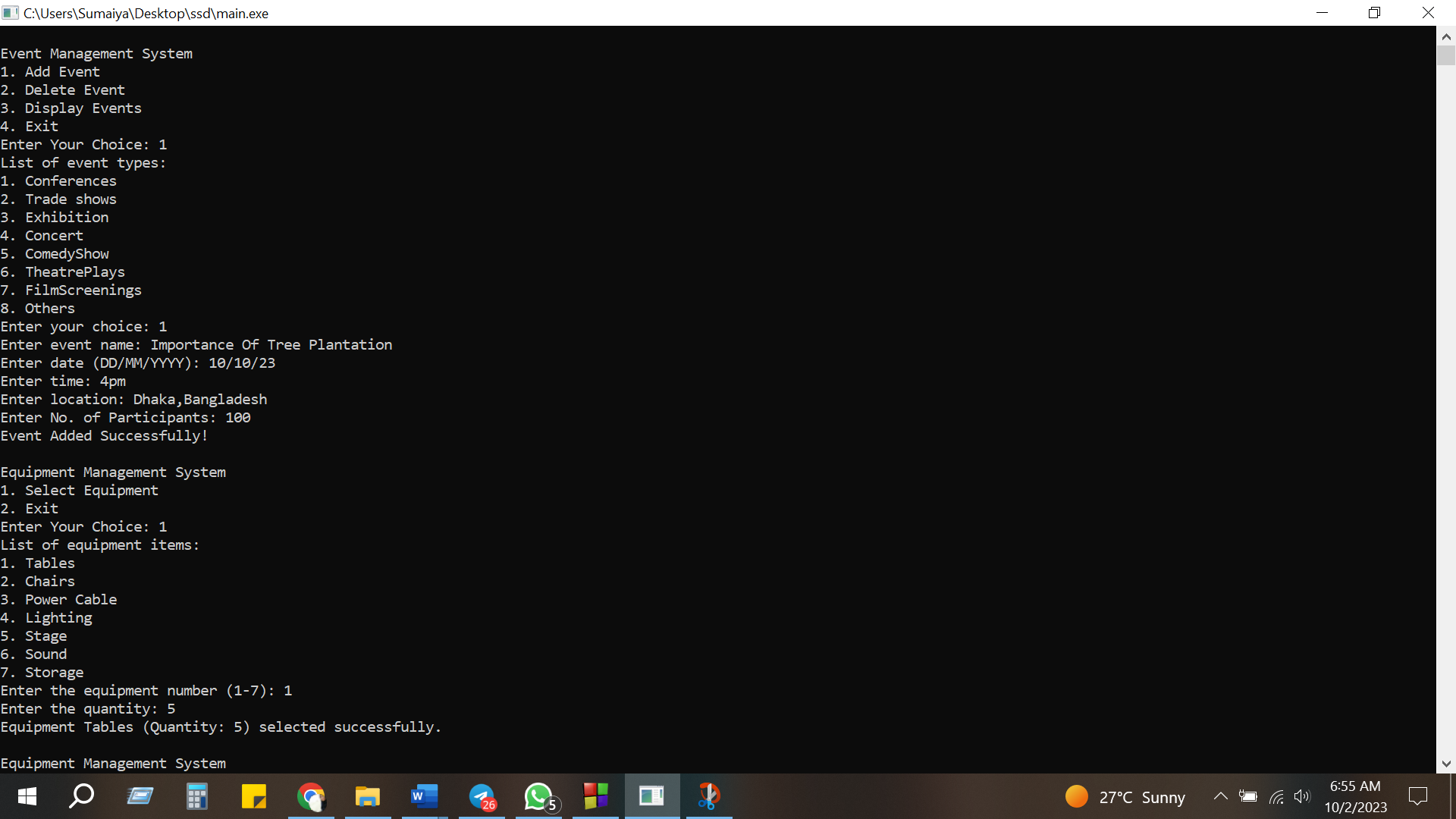
* ***Code Snippet:***

friend istream& operator>>(istream& input, EventDetails&event);

friend ostream& operator<<(ostream& output, EventDetails& event);

Main Function: The main function serves as the entry point of the program. It provides a user interface for interacting with the Event Management System. Users can add events of different types, select equipment for events, delete events, and display event details.

The code uses **polymorphism and dynamic memory allocation** to handle different event types and their details. It also tracks the cost of equipment selected for each event.Overall, this code provides a basic framework for managing events and equipment associated with those events. Users can interact with the system through a simple text-based menu.



CODE :

#include <bits/stdc++.h>

#include <iostream>

#include <vector>

#include <string>

using namespace std;

class Event

{

public:

virtual ~Event() {}

virtual void displayDetails() const = 0;

virtual void deleteEvent() = 0;

virtual double calculateCost() const = 0;

};

class EventType

{

public:

virtual ~EventType() {}

virtual void displayEventTypes() const = 0;

};

class Conference : public EventType

{

public:

void displayEventTypes() const override //polymorphism override function

{

cout << "1. Conferences" << endl;

}

};

class TradeShow : public EventType

{

public:

void displayEventTypes() const override

{

cout << "2. Trade shows" << endl;

}

};

class Exhibition : public EventType

{

public:

void displayEventTypes() const override

{

cout << "3. Exhibition" << endl;

}

};

class Concert : public EventType

{

public:

void displayEventTypes() const override

{

cout << "4. Concert" << endl;

}

};

class ComedyShow : public EventType

{

public:

void displayEventTypes() const override

{

cout << "5. ComedyShow" << endl;

}

};

class TheatrePlays : public EventType

{

public:

void displayEventTypes() const override

{

cout << "6. TheatrePlays" << endl;

}

};

class FilmScreenings : public EventType

{

public:

void displayEventTypes() const override

{

cout << "7. FilmScreenings" << endl;

}

};

class Others : public EventType

{

public:

void displayEventTypes() const override

{

cout << "8. Others" << endl;

}

};

class EventDetails : public Event

{

private:

string name;

string date;

string time;

string location;

int no\_of\_participants;

public:

EventDetails() {} //default constructor called

EventDetails(string n, string d, string t, string l, int participants) //parameterized constructor

: name(n), date(d), time(t), location(l), no\_of\_participants(participants){}

void displayDetails() const override

{

cout << "Event Details:" << endl;

cout << "Name: " << name << endl;

cout << "Date: " << date << endl;

cout << "Time: " << time << endl;

cout << "Location: " << location << endl;

cout << "No. of Participants: " << no\_of\_participants << endl;

}

friend istream& operator>>(istream& input, EventDetails& event);

friend ostream& operator<<(ostream& output, EventDetails& event);

void deleteEvent() override

{

}

double calculateCost() const override

{

double tablesCostPerUnit = 12;

double chairsCostPerUnit = 10;

double powerCableCost = 1000;

double lightingCost = 1500;

double stageCost = 2000;

double soundCost = 5000;

double storageCost = 1000;

double totalCost = tablesCostPerUnit + chairsCostPerUnit + powerCableCost + lightingCost + stageCost + soundCost + storageCost;

return totalCost \* no\_of\_participants;

}

};

class Equipment

{

public:

virtual ~Equipment() {}

virtual double getCost() const = 0;

};

// Derived classes for different equipment items

class Tables : public Equipment

{

public:

double getCost() const override

{

return 12.0;

}

};

class Chairs : public Equipment

{

public:

double getCost() const override

{

return 10.0;

}

};

class PowerCable : public Equipment

{

public:

double getCost() const override

{

return 1000.0;

}

};

class Lighting : public Equipment

{

public:

double getCost() const override

{

return 1500.0;

}

};

class Stage : public Equipment

{

public:

double getCost() const override

{

return 2000.0;

}

};

class Sound : public Equipment

{

public:

double getCost() const override

{

return 5000.0;

}

};

class Storage : public Equipment

{

public:

double getCost() const override

{

return 1000.0;

}

};

class EquipmentManager

{

private:

map<string, int> selectedEquipment; // Equipment name to quantity

public:

// Function to select equipment

void selectEquipment(const string &equipmentName, int quantity)

{

selectedEquipment[equipmentName] = quantity;

}

void displaySelectedEquipment() const

{

double totalCost = 0.0;

cout << "Selected Equipment:" << endl;

for (const auto &pair : selectedEquipment)

{

const string &equipmentName = pair.first;

int quantity = pair.second;

double costPerUnit = 0.0;

if (equipmentName == "Tables")

costPerUnit = 12.0;

else if (equipmentName == "Chairs")

costPerUnit = 10.0;

else if (equipmentName == "Power Cable")

costPerUnit = 1000.0;

else if (equipmentName == "Lighting")

costPerUnit = 1500.0;

else if (equipmentName == "Stage")

costPerUnit = 2000.0;

else if (equipmentName == "Sound")

costPerUnit = 5000.0;

else if (equipmentName == "Storage")

costPerUnit = 1000.0;

double equipmentCost = costPerUnit \* quantity;

totalCost += equipmentCost;

cout << equipmentName << ": " << quantity << " x $" << costPerUnit << " = $" << equipmentCost << endl;

}

cout << "Total Equipment Cost: $" << totalCost << endl;

}

};

class EventManager

{

private:

vector<Event \*> events;

public:

~EventManager()

{

for (Event \*event : events)

{

delete event;

}

}

//Function to Add event

void addEvent(Event \*event)

{

events.push\_back(event);

cout << "Event Added Successfully!" << endl;

}

//Function to Delete Event

void deleteEvent(int index)

{

if (index >= 0 && index < events.size())

{

delete events[index];

events.erase(events.begin() + index);

cout << "Event deleted successfully." << endl;

}

else

{

cout << "Invalid event index." << endl;

}

}

void displayEvents() const

{

cout << "List of Events:" << endl;

for (size\_t i = 0; i < events.size(); ++i)

{

cout << "Event " << i + 1 << ":" << endl;

events[i]->displayDetails();

cout << "Cost: $" << events[i]->calculateCost() << endl;

cout << "-----------------------" << endl;

}

}

};

istream& operator>>(istream& input, EventDetails& event)

{

cout << "Enter event name: ";

input.ignore();

getline(input, event.name);

cout << "Enter date (DD/MM/YYYY): ";

input >> event.date;

cout << "Enter time: ";

input >> event.time;

cout << "Enter location: ";

input.ignore();

getline(input, event.location);

cout << "Enter No. of Participants: ";

input >> event.no\_of\_participants;

return input;

}

int main()

{

EventManager manager;

Conference conference;

TradeShow tradeshow;

Exhibition exhibition;

Concert concert;

ComedyShow comedyshow;

TheatrePlays theatreplays;

FilmScreenings filmscreenings;

Others others;

EquipmentManager equipmentManager;

int choice,choicce;

char exit;

do

{

cout << "\nEvent Management System" << endl;

cout << "1. Add Event" << endl;

cout << "2. Delete Event" << endl;

cout << "3. Display Events" << endl;

cout << "4. Exit" << endl;

cout << "Enter Your Choice: ";

cin >> choice;

switch (choice)

{

case 1:

{

cout << "List of event types:" << endl;

conference.displayEventTypes();

tradeshow.displayEventTypes();

exhibition.displayEventTypes();

concert.displayEventTypes();

comedyshow.displayEventTypes();

theatreplays.displayEventTypes();

filmscreenings.displayEventTypes();

others.displayEventTypes();

cout << "Enter your choice: ";

int evchoice;

cin >> evchoice;

Event \*newEvent = nullptr;

if (evchoice == 1 || evchoice == 2 || evchoice == 3 || evchoice == 4 || evchoice == 5 || evchoice == 6 || evchoice == 7 || evchoice == 8)

{

EventDetails new\_event;

cin>>new\_event;

Event\* newEventPtr = new EventDetails(new\_event);

EventDetails(new\_event);

manager.addEvent(newEventPtr);

}

do

{

cout << "\nEquipment Management System" << endl;

cout << "1. Select Equipment" << endl;

cout << "2. Exit" << endl;

cout << "Enter Your Choice: ";

cin >> choicce;

switch (choicce)

{

case 1:

{

cout << "List of equipment items:" << endl;

cout << "1. Tables" << endl;

cout << "2. Chairs" << endl;

cout << "3. Power Cable" << endl;

cout << "4. Lighting" << endl;

cout << "5. Stage" << endl;

cout << "6. Sound" << endl;

cout << "7. Storage" << endl;

cout << "Enter the equipment number (1-7): ";

int equipmentChoice;

cin >> equipmentChoice;

if (equipmentChoice >= 1 && equipmentChoice <= 7)

{

string equipmentName;

int quantity;

switch (equipmentChoice)

{

case 1:

equipmentName = "Tables";

break;

case 2:

equipmentName = "Chairs";

break;

case 3:

equipmentName = "Power Cable";

break;

case 4:

equipmentName = "Lighting";

break;

case 5:

equipmentName = "Stage";

break;

case 6:

equipmentName = "Sound";

break;

case 7:

equipmentName = "Storage";

break;

default:

cout<<"nothing here"<<endl;

}

cout << "Enter the quantity: ";

cin >> quantity;

equipmentManager.selectEquipment(equipmentName, quantity);

cout << "Equipment " << equipmentName << " (Quantity: " << quantity << ") selected successfully." << endl;

}

else

{

cout << "Invalid equipment choice. Please try again." << endl;

}

break;

}

case 2:

{

cout << "Exiting equipment management." << endl;

break;

}

default:

cout << "Invalid choice. Please try again." << endl;

}

}

while(choicce!=2);

break;

}

case 2:

{

cout << "Enter the index of the event you want to delete: ";

int index;

cin >> index;

manager.deleteEvent(index - 1);

break;

}

case 3:

{

manager.displayEvents();

break;

}

case 4:

{

cout << "Exiting program." << endl;

}

default:

cout << "Invalid choice. Please try again." << endl;

break;

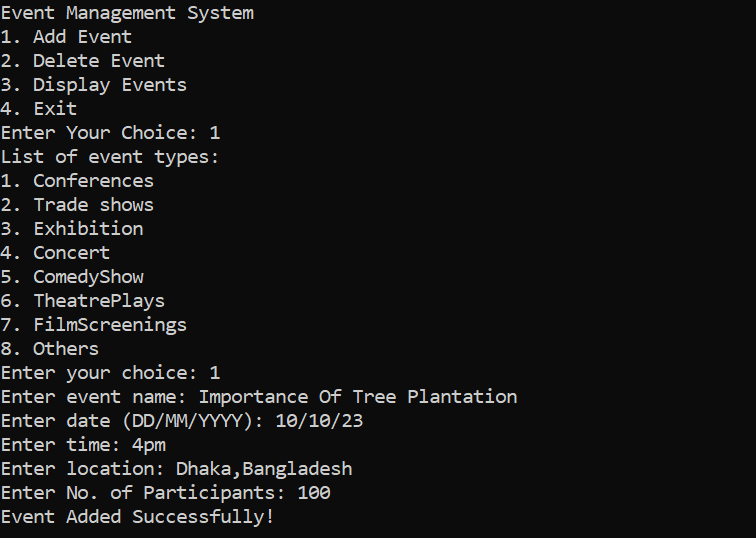
}

}

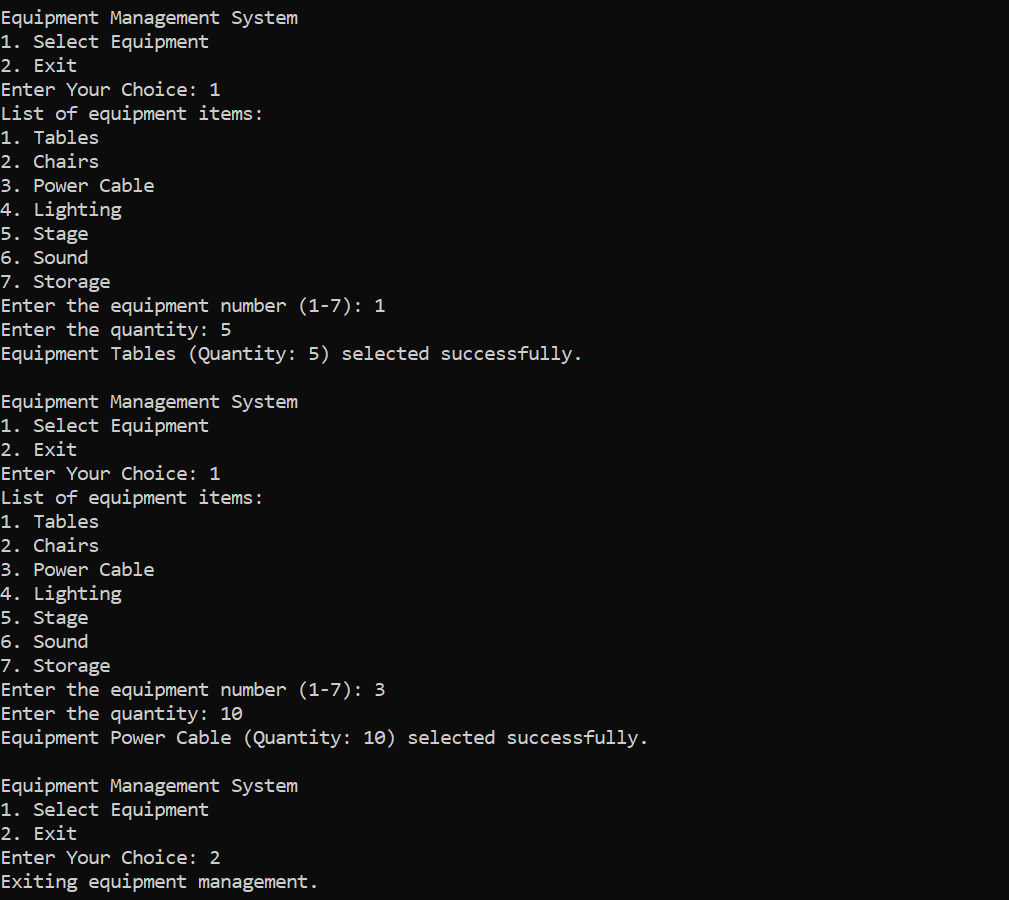
while(choice!=4);

return 0;

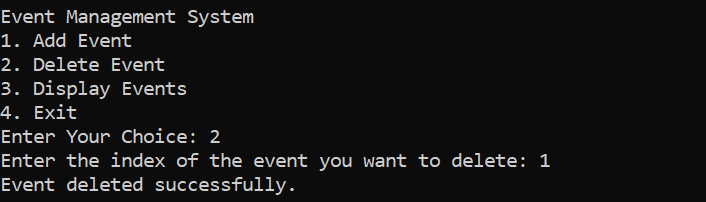
}

**Output: *ADD EVENT***

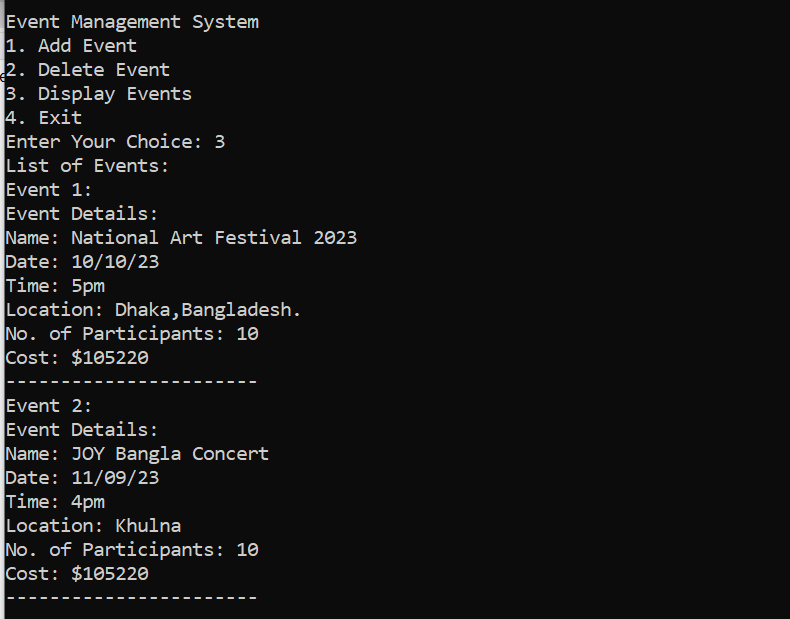
***EQUIPMENT MANAGEMENT SYSTEM***

****

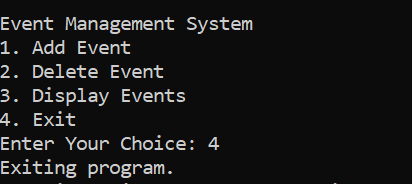
***DELETE EVENT***



***DISPLAY EVENTS***



***EXIT***

******

**Explanation of Code:**

This code is an example of an Event Management System implemented in C++ and demonstrates various Object-Oriented Programming (OOP) concepts. Let's break down the code and explain how it incorporates these concepts:Classes and Objects:**Classes** are used to model different entities in the system, such as Event, EventType, EventDetails, Equipment, Tables, Chairs, etc.**Objects** of these classes are created in the main function. For example, objects like EventDetails, Conference, EventManager, and EquipmentManager are instantiated.

Inheritance:Inheritance is used to establish relationships between classes.EventDetails is derived from the base class Event.EventType, Conference, TradeShow, and other classes demonstrate inheritance as they are derived from the base class EventType.

Polymorphism:The code uses runtime polymorphism by defining virtual functions in the Event and EventType base classes and providing different implementations in their derived classes.The displayDetails, deleteEvent, and calculateCost functions in the Event class are declared as pure virtual functions, allowing derived classes to override them.

Encapsulation:Encapsulation is achieved by using private and public access specifiers.Private member variables (e.g., name, date, time, etc.) in the EventDetails class are encapsulated, and their access is controlled through member functions.

Abstraction:Abstraction is the concept of hiding the internal details and showing only the necessary functionalities.The abstract base classes Event and EventType provide a level of abstraction by defining the interfaces for derived classes.

Operator Overloading:Operator overloading is used to define custom behavior for operators like << and >>.In this code, operator<< is overloaded for the EventDetails class to allow for customized output, and operator>> is overloaded for input.

Constructor Overloading:Constructor overloading is demonstrated in the EventDetails class.It provides both a default constructor and a parameterized constructor to initialize objects.

Standard Template Library (STL):The code uses various STL components like vector and map.**A vector** is used to manage a collection of Event\* objects in the EventManager class.A **map** is used in the EquipmentManager class to keep track of selected equipment items.

Dynamic Memory Allocation and Deallocation:Dynamic memory allocation is used when creating new event objects with new keyword in the addEvent function.Dynamic memory deallocation is done in the destructor of the EventManager class to prevent memory leaks.

Overall, this code demonstrates key OOP principles such as inheritance, encapsulation, polymorphism, and abstraction. It also leverages C++ features like operator overloading and the STL for efficient event and equipment management.

**Future Enhancements:**

The provided Event Management System is a basic implementation, and there are several ways to enhance its functionality and usability. Here are some future enhancements and features that can be added to improve the system:User Authentication:Implement user authentication to allow multiple users to log in with different access levels (e.g., admin, event organizer, participant).

Event Scheduling:Add the ability to schedule events for specific dates and times. Implement a calendar view to display events on a calendar.

Event Registration:Allow participants to register for events online. Send confirmation emails to participants upon successful registration.

Payment Processing:Integrate payment gateways to handle event registration fees. Provide online payment options for participants.

Event Categories and Tags:Categorize events based on types, themes, or tags to make it easier for users to find events of interest.

Event Ratings and Reviews:Allow participants to rate and review events. Display ratings and reviews to help users make informed decisions.

Event Notifications:Send event notifications and reminders to participants via email or SMS.Allow users to subscribe to event updates.

Event Analytics:Implement analytics to track event attendance, user engagement, and other relevant metrics. Provide event organizers with insights to improve future events.

Venue Management:Create a venue management system to manage event locations, including booking and availability.

QR Code Check-In:

Generate QR codes for event participants for easy check-in at the venue.

Event Surveys and Feedback:Conduct post-event surveys to gather feedback from participants. Use feedback to enhance future events.

Event Collaboration:Allow multiple event organizers to collaborate on planning and managing events. Share event details and responsibilities among organizers.

Mobile Application:Develop a mobile app for the Event Management System to increase accessibility and convenience for users.

Localization:Add multi-language support to cater to a broader audience.

Data Backup and Security:Implement regular data backups and security measures to protect user and event data.

Customizable Event Pages:Allow event organizers to create customized event pages with branding and design options.

Social Media Integration:Enable users to share events on social media platforms. Integrate with social media APIs for event promotion.

Export and Reporting:Provide tools for event organizers to export attendee lists and generate event reports.

Integration with External Services:Integrate with external services like weather updates, transportation information, and local attractions for event planning.

Feedback and Feature Requests:Implement a feedback system where users can submit feature requests and report issues.Use user feedback to prioritize future enhancements. These enhancements would transform the basic Event Management System into a more comprehensive and user-friendly platform for event planning, promotion, and organization. The specific features to prioritize will depend on the target audience and the system's intended use cases.

**Discussion:**

The provided code implemes an Event Management System in C++ which allows users to add, delete, and display various types of events, along with selecting equipment for those events. The code leverages polymorphism and inheritance through the use of base classes like Event, EventType, and Equipment, along with their respective derived classes. This allows for flexibility in managing different types of events and equipment, making it easy to extend the system in the future by adding new event types or equipment. The Event class serves as the base class for event-related operations, while the Equipment class handles equipment-related operations. The derived classes, such as EventDetails, Tables, Chairs, etc., provide concrete implementations for specific event details and equipment types. This structure promotes code reusability and organization.

The system efficiently handles user input and output. It uses a menu-driven approach, enabling users to interact with the program by selecting options from the menu. The istream and ostream operator overloads facilitate user-friendly input and output for event details.The EventManager class manages events, allowing users to add and delete events. Events are stored as pointers to the base class, facilitating dynamic polymorphism. The EquipmentManager class manages equipment selection, keeping track of the chosen equipment items and their costs.The calculateCost method in the EventDetails class calculates the total cost of an event based on the number of participants and fixed cost values for tables, chairs, power cables, lighting, stage, sound, and storage. This feature is beneficial for event planning and budgeting. The program uses vectors and maps to store events and selected equipment, respectively. This data storage approach ensures efficient retrieval and management of information.

**Conclusion:**

The provided Event and Equipment Management System demonstrates the use of object-oriented programming principles such as polymorphism, inheritance, and encapsulation to create a well-structured and extensible application. It allows users to interactively manage events of various types and select equipment items, making it suitable for event planning and organization. The code lacks robust error handling for user input. It should include input validation to prevent invalid data entry. The code uses raw pointers to manage events, which may lead to memory leaks if not handled carefully. Consider using smart pointers (e.g., std::shared\_ptr) for safer memory management. The user interface could be enhanced with more informative messages and better guidance for users. While the current system supports various event types and equipment items, it may benefit from a more modular approach to easily add new event types and equipment in the future.Overall, this Event and Equipment Management System serves as a solid foundation for event planning and organization, and with some enhancements, it could become a valuable tool for managing a wide range of events efficiently.