Name: Zhongtain Dai

Student ID: 335953

Preliminary Project “Sport Club”

for my project "Sport Club," which involves designing a system to manage various aspects of a sport club using object-oriented programming principles, I will outline five classes with their respective functionalities and relationships.

Classes Overview

1. **Club**
    - The main class that connected with other classes.
    - It Manages overall club operations including adding/removing members, coaches, teams, and organizing events.
2. **Member**
    - Represents club members, both athletes and possibly non-athletic members.
    - It Manages personal information and roles within the club.
3. **Coach**
    - Manages coaches who train the club's athletes.
    - It Includes functionalities for assigning coaches to teams and planning training sessions.
4. **Team**
    - Represents different sports teams within the club.
    - Manages team members, assigns coaches, and handles team-specific operations.
5. **Event**
    - Manages events like matches, tournaments, or training sessions.
    - Includes functionalities for scheduling, rescheduling, and managing participants.

Class Descriptions and Functionalities

1. Club

- **Attributes:**
    - **name**: the name of the sports club.
    - **members**: a list of members in the club.
    - **coaches**: a list of coaches.
    - **teams**: a list of sports teams.
    - **events**: a list of scheduled events, including matches, tournaments, and training sessions, managed by the club.

- **Methods:**
    - **addMember:** Add a new member to the club.
    - **removeMember:** Remove a member from the club.(it means we also removed it from team , event)
    - **addCoach**(std::shared_ptr<Coach> coach): Add a new coach.
    - **removeCoach**(std::shared_ptr<Coach> coach): Remove a coach from the club.
    - **addTeam:** Create a new sports team.
    - **removeTeam:** Remove a sports team.
    - **organizeEvent:** Schedule a new event.
    - **cancelEvent:** Cancel an event.
    - **addMembersToEvent:** Add members to a specific event.
    - `addTeamToEvent: add teams to event.`
    - **findMemberByName const:** Find a member by name.
    - **findMembersByRole:** Find members by role.
    - **findCoachByName const:** Find a coach by name.
    - `findMemberById:` Searches for and returns a member object with the specified ID.
    - `findCoachById:` Searches for and returns a coach object with the specified ID.
    - `findPersonById:` Searches for and displays the details of a person (either a member or a coach) with the specified ID.
    - **updateCoachSpecialty:** Update a coach's specialty.
    - **hasScheduleConflict:** Check for schedule conflicts.

    Get data from private class :

    - **getClubInfo() const:** Get information about the club.
    - **getMembers() const:** Get the list of club members.
    - **getCoaches() const:** Get the list of club coaches.
    - **getTeams() const:** Get the list of club teams.
    - **getEvents() const:** Get the list of club events.
- ~**Club()**;:Destructors delete club(if we delete club the member, team, event, coach in this club will be also deleted

- `class Club {`
- `private:`

```
    std::string name;
    std::vector<Member*> members;
    std::vector<Coach*> coaches;
    std::vector<Team*> teams;
    std::vector<Event*> events;

public:
    explicit Club(const std::string& name);

    ~Club();

    void addMember(Member* member);
    void removeMember(Member* member);
    void addCoach(Coach* coach);
    void removeCoach(Coach* coach);
    void addTeam(Team* team);
    void removeTeam(Team* team);
    void organizeEvent(Event* event);
    void cancelEvent(Event* event);
    void addMembersToEvent(const std::string& eventName, const
std::vector<Member*>& newMembers);
void addTeamToEvent(const std::string& eventName, Team* team);

    Member* findMemberByName(const std::string& name) const;
    std::vector<Member*> findMembersByRole(const std::string& role) const;
    Coach* findCoachByName(const std::string& name) const;
    void updateCoachSpecialty(const std::string& name, const std::string&
new_specialty);
    bool hasScheduleConflict(const std::string& date) const;

    Member* findMemberById(int id) const;
    Coach* findCoachById(int id) const;
    void findPersonById(int id) const;

    std::string getClubInfo() const;
    std::vector<Member*> getMembers() const;
    std::vector<Coach*> getCoaches() const;
    std::vector<Team*> getTeams() const;
    std::vector<Event*> getEvents() const;
};

#endif // CLUB_H
```

Club Class Tests

Correct Cases

1. ***Adding and Updating Members***
    - ***Description: Add a member, update details, and verify updates.***
2. ***Adding and Removing Coach***
    - ***Description: Add a coach, remove the coach, and verify changes.***
3. ***Organizing and Cancelling Event***
    - ***Description: Organize an event, add participants, and cancel the event.***

4. ***Successfully delete club and all associated objects***

Incorrect Cases

1. ***Adding Member with Empty Name***
    - ***Description: Add a member with an empty name.***
    - ***Outcome: Throws invalid_argument for empty name.***
2. ***Adding Coach with Empty Specialty***
    - ***Description: Add a coach with an empty specialty.***
    - ***Outcome: Throws invalid_argument for empty specialty.***
3. ***Organizing Event with Empty Date***
    - ***Description: Organize an event with an empty date.***
    - ***Outcome: Throws invalid_argument for empty date.***
4. ***Adding with duplicate data for coach and member: can not add same id***

```
void testClub() {
    try {
        Club club("Sports Club");

        // Adding and updating member
        Member* m1 = new Member("John", 30, "Athlete", 1);
        Member* m2 = new Member("Jane", 25, "Athlete", 2);
        club.addMember(m1);
        assert(club.getMembers().size() == 1);

        m1->updateDetails("John D.", 31);
        assert(m1->getName() == "John D.");
        assert(m1->getAge() == 31);

        club.removeMember(m1);
        assert(club.getMembers().size() == 0);
        delete m1;

        // Adding and removing coach
        Coach* c1 = new Coach("Jane", "Football", 1);
        club.addCoach(c1);
        assert(club.getCoaches().size() == 1);
        club.removeCoach(c1);
        assert(club.getCoaches().size() == 0);
        delete c1;

        // Organizing and cancelling event
        Event* e1 = new Event("2024-04-19", "Stadium", "Football Match");
        club.organizeEvent(e1);
        assert(club.getEvents().size() == 1);
        club.cancelEvent(e1);
        assert(club.getEvents().size() == 0);
```

```
        delete e1;

        // Re-create objects to test event participation
        e1 = new Event("2024-04-19", "Stadium", "Football Match");
        club.organizeEvent(e1);
        club.addMember(m2);
        std::vector<Member*> members = { m2 };
        club.addMembersToEvent("Football Match", members);
        assert(club.getEvents().front()->getParticipants().size() == 1);

        c1 = new Coach("Jane", "Football", 1);
        club.addCoach(c1);
        Coach* foundCoach = club.findCoachByName("Jane");
        assert(foundCoach != nullptr);
        assert(foundCoach->getSpecialty() == "Football");

        club.updateCoachSpecialty("Jane", "Basketball");
        assert(foundCoach->getSpecialty() == "Basketball");

        // Testing duplicate member addition
        try {
            Member* m3 = new Member("Jane", 25, "Athlete", 2); // Duplicate ID
            club.addMember(m3);
            std::cerr << "testDuplicateMember failed: no exception on duplicate
member ID" << std::endl;
        }
        catch (const std::invalid_argument& e) {
            std::cout << "Caught expected exception for duplicate member ID: " <<
e.what() << std::endl;
        }

        // Testing duplicate coach addition
        try {
            Coach* c2 = new Coach("Jane", "Football", 1); // Duplicate ID
            club.addCoach(c2);
            std::cerr << "testDuplicateCoach failed: no exception on duplicate
coach ID" << std::endl;
        }
        catch (const std::invalid_argument& e) {
            std::cout << "Caught expected exception for duplicate coach ID: " <<
e.what() << std::endl;
        }

        std::cout << "testClub passed" << std::endl;

        delete m2;
        delete c1;
        delete e1;
    }
    catch (...) {
        std::cout << "testClub failed" << std::endl;
    }
}}


void testDeleteClub() {
    try {
        std::cout << "Starting testDeleteClub" << std::endl;
        Club* myClub = new Club("Sports Club");

        std::cout << "Adding members and coaches" << std::endl;
        Member* m1 = new Member("John", 30, "Athlete", 1);
        Member* m2 = new Member("Jane", 25, "Athlete", 2);
```

```
        myClub->addMember(m1);
        myClub->addMember(m2);

        Coach* c1 = new Coach("Coach A", "Football", 1);
        myClub->addCoach(c1);

        std::cout << "Organizing event" << std::endl;
        Event* e1 = new Event("2024-04-19", "Stadium", "Football Match");
        myClub->organizeEvent(e1);

        std::cout << "Adding team" << std::endl;
        Team* t1 = new Team("Football", c1, 1);
        t1->addMember(m1);
        t1->addMember(m2);
        myClub->addTeam(t1);

        // Deleting club and all associated objects
        std::cout << "Deleting club" << std::endl;
        delete myClub;
        std::cout << "testDeleteClub passed" << std::endl;
    }
    catch (const std::exception& e) {
        std::cerr << "testDeleteClub failed: " << e.what() << std::endl;
    }
    catch (...) {
        std::cerr << "testDeleteClub failed: unknown exception" << std::endl;
    }
}

void testDeleteClub() {
    try {
        std::cout << "Starting testDeleteClub" << std::endl;
        Club* myClub = new Club("Sports Club");

        std::cout << "Adding members and coaches" << std::endl;
        Member* m1 = new Member("John", 30, "Athlete", 1);
        Member* m2 = new Member("Jane", 25, "Athlete", 2);
        myClub->addMember(m1);
        myClub->addMember(m2);

        Coach* c1 = new Coach("Coach A", "Football", 1);
        myClub->addCoach(c1);

        std::cout << "Organizing event" << std::endl;
        Event* e1 = new Event("2024-04-19", "Stadium", "Football Match");
        myClub->organizeEvent(e1);

        std::cout << "Adding team" << std::endl;
        Team* t1 = new Team("Football", c1, 1);
        t1->addMember(m1);
        t1->addMember(m2);
        myClub->addTeam(t1);

        // Deleting club and all associated objects
        std::cout << "Deleting club" << std::endl;
        delete myClub;
        std::cout << "testDeleteClub passed" << std::endl;
    }
    catch (const std::exception& e) {
        std::cerr << "testDeleteClub failed: " << e.what() << std::endl;
    }
    catch (...) {
        std::cerr << "testDeleteClub failed: unknown exception" << std::endl;
```

```
    }
}
```

2. Member

- **Attributes:**
    - **id**: a unique identifier for the member.
    - **name**: the member's name.
    - **age**: the member's age.
    - **role**: distinguishes between athlete, staff, etc.
- **Methods:**
    - **update_details()**: Updates the member's personal details.
    - **Operator ==** : compare class data between 2 member
    - **Other method to get data from private class:**
    - **int getId() const**: Retrieves the member's ID.
    - **std::string getName() const**: Retrieves the member's name.
    - **int getAge() const**: Retrieves the member's age.
    - **std::string getRole() const**: Retrieves the member's role.

```
class Member {
private:
    std::string name;
    int age;
    std::string role;
    int id;

public:
    Member(const std::string& name, int age, const std::string& role, int id);

    std::string getName() const;
    int getAge() const;
    std::string getRole() const;
    void updateDetails(const std::string& new_name, int new_age);
    int getId() const;
    bool operator==(const Member& other) const;
};
```

Member Class Tests

> ***Correct Creation Test:*** *This test ensures that when a Member object is created with valid parameters, the attributes are set as expected. It creates a Member with a name, age, and role, and then asserts that these attributes match the provided values.*
>
> ***Update Details Test (Correct)****:This test verifies that the updateDetails method correctly updates the member's name and age. After updating, it asserts that both the name and age match the new values.*

***Update Details Test (Incorrect)**: method when passed invalid values, such as an empty name or negative age. It attempts to update the member with invalid values and expects an exception to be thrown.*

Test id : id can not be negative

Test member : removing a member and its effect on associated teams and events(correct case)

```
#include "Member.h"

void testMember() {
    try {
        Member* m1 = new Member("John", 30, "Athlete", 1);

        assert(m1->getName() == "John");
        assert(m1->getAge() == 30);
        assert(m1->getRole() == "Athlete");
        assert(m1->getId() == 1); // Check ID

        m1->updateDetails("John D.", 31);
        assert(m1->getName() == "John D." && m1->getAge() == 31);

        try {
            m1->updateDetails("", 31);
            std::cerr << "testMember failed: no exception on empty name" <<
std::endl;
        }
        catch (const std::invalid_argument& e) {
            std::cout << "Caught expected exception for empty name: " << e.what()
<< std::endl;
        }

        try {
            m1->updateDetails("John D.", -5);
            std::cerr << "testMember failed: no exception on negative age" <<
std::endl;
        }
        catch (const std::invalid_argument& e) {
            std::cout << "Caught expected exception for negative age: " <<
e.what() << std::endl;
        }


        try {
            Member* m2 = new Member("Jane", 25, "Athlete", -1);
            std::cerr << "testMember failed: no exception on negative ID" <<
std::endl;
            delete m2;
        }
        catch (const std::invalid_argument& e) {
            std::cout << "Caught expected exception for negative ID: " << e.what()
<< std::endl;
        }


        std::cout << "testMember passed" << std::endl;

        delete m1;
    }
    catch (const std::exception& e) {
        std::cerr << "testMember failed: " << e.what() << std::endl;
```

```
    }
    catch (...) {
        std::cerr << "testMember failed: unknown exception" << std::endl;
    }
}

void testRemoveMember() {
    std::cout << "Starting testRemoveMember" << std::endl;
    Club myClub("Sports Club");

    Member* m1 = new Member("John", 30, "Athlete", 1);
    Member* m2 = new Member("Jane", 25, "Athlete", 2);
    myClub.addMember(m1);
    myClub.addMember(m2);

    Coach* c1 = new Coach("Coach A", "Football", 1);
    myClub.addCoach(c1);

    Event* e1 = new Event("2024-04-19", "Stadium", "Football Match");
    myClub.organizeEvent(e1);
    myClub.addMembersToEvent("Football Match", { m1, m2 });

    Team* t1 = new Team("Football", c1, 1);
    t1->addMember(m1);
    t1->addMember(m2);
    myClub.addTeam(t1);

    std::cout << "Finding person with ID 1 before deletion:" << std::endl;
    myClub.findPersonById(1);

    std::cout << "Removing member with ID 1:" << std::endl;
    myClub.removeMember(m1);

    std::cout << "Finding person with ID 1 after deletion:" << std::endl;
    myClub.findPersonById(1);

    // Check if the member is removed from the team
    if (t1->getMembers().size() == 1 && t1->getMembers()[0]->getId() == 2) {
        std::cout << "Member successfully removed from team." << std::endl;
    }
    else {
        std::cerr << "Failed to remove member from team." << std::endl;
    }

    // Check if the member is removed from the event
    if (e1->getParticipants().size() == 1 && e1->getParticipants()[0]->getId() ==
2) {
        std::cout << "Member successfully removed from event." << std::endl;
    }
    else {
        std::cerr << "Failed to remove member from event." << std::endl;
    }

    std::cout << "testRemoveMember completed" << std::endl;
}
```

3. coach

Attributes:

- id: A unique identifier for the coach.
- name: The name of the coach.

- specialty: The specialty or area of expertise of the coach.

Methods:
- int getId() const: Retrieves the coach's unique identifier.
- std::string getName() const: Retrieves the coach's name.
- std::string getSpecialty() const: Retrieves the coach's specialty.
- void setSpecialty(const std::string& new_specialty): Updates the coach's specialty.
- bool operator==(const Coach& other) const: Compares two coach objects for equality based on their id, name, and specialty.

```
#include <string>

class Coach {
private:
    std::string name;
    std::string specialty;
    int id;

public:
    Coach(const std::string& name, const std::string& specialty, int id);

    std::string getName() const;
    std::string getSpecialty() const;
    void setSpecialty(const std::string& new_specialty);
    int getId() const;
    bool operator==(const Coach& other) const;
      };
```

Coach Class Tests

1. **Correct Creation Test**: Similar to the Member class, this test creates a Coach object with an ID, name, and specialty, and then verifies that these attributes are correctly assigned.
2. **Specialty Assignment Test (Correct):** This test confirms that the Coach's specialty is correctly set upon creation.
3. **Specialty Assignment Test (Incorrect):** This would test how the class behaves when a coach is created without a specialty. The assumption here is that a valid coach must have a specialty, and hence the test would fail or trigger an exception if a coach is created with an empty specialty. This test is hypothetical and assumes that such validation logic exists.
4. **Test id :** id cannot be negative
5. **Test coach :** removing a coach and its effect on associated teams(correct case)

```
void testCoach() {
    try {
        // Correct Creation Test
        Coach* c1 = new Coach("Jane", "Football", 1);
        assert(c1->getName() == "Jane");
        assert(c1->getSpecialty() == "Football");
        assert(c1->getId() == 1); // Check ID

        // Specialty Assignment Test (Correct)
        c1->setSpecialty("Basketball");
        assert(c1->getSpecialty() == "Basketball");

        // Specialty Assignment Test (Incorrect)
        try {
            Coach* c2 = new Coach("John", "", 2);
```

```
            std::cerr << "testCoach failed: no exception on empty specialty" <<
std::endl;
            delete c2;  // Ensure to delete c2 if no exception is thrown
        }
        catch (const std::invalid_argument& e) {
            std::cout << "Caught expected exception for empty specialty: " <<
e.what() << std::endl;
        }
        try {
            Coach* c3 = new Coach("John", "Football", -1);
            std::cerr << "testCoach failed: no exception on negative ID" <<
std::endl;
            delete c3;
        }
        catch (const std::invalid_argument& e) {
            std::cout << "Caught expected exception for negative ID: " << e.what()
<< std::endl;
        }

        std::cout << "testCoach passed" << std::endl;

        delete c1;
    }
    catch (...) {
        std::cout << "testCoach failed" << std::endl;
    }
      }

void testRemoveCoach() {
    std::cout << "Starting testRemoveCoach" << std::endl;
    Club myClub("Sports Club");

    Member* m1 = new Member("John", 30, "Athlete", 1);
    Member* m2 = new Member("Jane", 25, "Athlete", 2);
    myClub.addMember(m1);
    myClub.addMember(m2);

    Coach* c1 = new Coach("Coach A", "Football", 1);
    myClub.addCoach(c1);

    Event* e1 = new Event("2024-04-19", "Stadium", "Football Match");
    myClub.organizeEvent(e1);
    myClub.addMembersToEvent("Football Match", { m1, m2 });

    Team* t1 = new Team("Football", c1, 1);
    t1->addMember(m1);
    t1->addMember(m2);
    myClub.addTeam(t1);

    std::cout << "Finding person with ID 1 (Coach) before deletion:" << std::endl;
    myClub.findPersonById(1);

    std::cout << "Removing coach with ID 1:" << std::endl;
    myClub.removeCoach(c1);

    std::cout << "Finding person with ID 1 (Coach) after deletion:" << std::endl;
    myClub.findPersonById(1);

    // Check if the team still exists but without a coach
    if (t1->getCoach() == nullptr) {
        std::cout << "Coach successfully removed from team, team still exists."
<< std::endl;
    }
```

```
    else {
        std::cerr << "Failed to remove coach from team." << std::endl;
    }

    std::cout << "testRemoveCoach completed" << std::endl;
      }
```

4. Team

- **Attributes:**
    - **team_id**: unique identifier for the team.
    - **sport_type**: type of sport the team plays.
    - **members**: list of members who are part of the team.
    - **coach**: the coach assigned to the team.
- **Operators**
    - **Operator +** : to merge 2 team into a new team objetct
    - **Bool operator ==**: compare 2 team data
    - **Operator =** : to assign all member values from one team to another
    -
- **Methods:**
    - ***addMember(Member)**: Adds a member to the team.**
    - ***removeMember(Member)**: Removes a member from the team.**
    - `removeCoach: removes coach (as it is possible that teams can exist without coach)`
    - **getTeamId() const: Retrieves the team's unique identifier.**
    - **getSportType() const: Retrieves the type of sport the team plays.**
    - **getMembers() const: Retrieves the list of team members.**
    - **getCoach() const: Retrieves the coach assigned to the team.**
    - **getid()**-get the id

```
#include <vector>
#include <string>
#include "Member.h"
#include "Coach.h"

class Team {
private:
    std::string sport_type;
    std::vector<Member*> members;
    Coach* coach;
    int id;

public:
    Team(const std::string& sport_type, Coach* coach, int id);
```

```
    void addMember(Member* member);
    void removeMember(Member* member);
    void setCoach(Coach* coach);

    std::string getSportType() const;
    std::vector<Member*> getMembers() const;
    Coach* getCoach() const;
     void removeCoach();

    bool operator==(const Team& other) const;
    Team operator+(const Team& other) const;
    int getId() const;
    size_t getMemberCount() const;
};
```

Team Class Tests

1. **Adding Member Test (Correct):** This test ensures that when a Member is added to a Team, the Team's member count is incremented correctly.
2. **Removing Member Test (Correct):** After adding a member, this test checks if the removeMember method successfully removes the Member from the Team, and the member count is decremented.
3. **Removing Non-Existent Member Test (Incorrect):** This test would check the behavior of the removeMember method when attempting to remove a Member who is not part of the Team. The test is to ensure that the member count remains unchanged.
4. **Test id :** id can not be negative

```
void testTeam() {
    // Adding Member Test (Correct)
    try {
        Coach* coach = new Coach("Jane", "Football", 1);
        Team team("Football", coach, 1);

        Member* member1 = new Member("John", 30, "Athlete", 1);
        team.addMember(member1);

        assert(team.getMembers().size() == 1);
        std::cout << "testAddingMember passed" << std::endl;

        delete member1;
        delete coach;
    }
    catch (...) {
        std::cout << "testAddingMember failed" << std::endl;
    }

    // Removing Member Test (Correct)
    try {
        Coach* coach = new Coach("Jane", "Football", 1);
        Team team("Football", coach, 1);

        Member* member1 = new Member("John", 30, "Athlete", 1);
        team.addMember(member1);
        assert(team.getMembers().size() == 1);

        team.removeMember(member1);
        assert(team.getMembers().size() == 0);
        std::cout << "testRemovingMember passed" << std::endl;
```

```
        delete member1;
        delete coach;
    }
    catch (...) {
        std::cout << "testRemovingMember failed" << std::endl;
    }

    // Removing Non-Existent Member Test (Incorrect)
    try {
        Coach* coach = new Coach("Jane", "Football", 1);
        Team team("Football", coach, 1);

        Member* member1 = new Member("John", 30, "Athlete", 1);
        Member* member2 = new Member("Jane", 25, "Athlete", 2);

        team.addMember(member1);
        assert(team.getMembers().size() == 1);

        team.removeMember(member2); // member2 was never added
        assert(team.getMembers().size() == 1); // Size should remain unchanged
        std::cout << "testRemovingNonExistentMember passed" << std::endl;



        delete member1;
        delete member2;
        delete coach;
    }
    catch (...) {
        std::cout << "testRemovingNonExistentMember failed" << std::endl;
    }

    try {
        Coach* coach = new Coach("Jane", "Football", 1);
        Team* team = new Team("Football", coach, -1);  // Negative ID
        std::cerr << "testTeam failed: no exception on negative ID" << std::endl;
        delete team;
        delete coach;
    }
    catch (const std::invalid_argument& e) {
        std::cout << "Caught expected exception for negative ID: " << e.what() <<
std::endl;
    }
}
```

5. Event

- **Attributes:**
    - **event_id**: a unique identifier for the event.
    - **date**: the date of the event.
    - **location**: where the event is taking place.
    - **participants**: list of participating members or teams.
- **Methods:**

- **reschedule(const std::string& date): Changes the date of the event.**
- ***addParticipant(Member participant)**:** Adds a participant (member) to the event.
- ***addParticipant(Team team)**:** Adds a participant (team) to the event.
- **bool operator==(const Event& other) const:** Compares two events for equality based on **their data.**
- **getDate() const:** Retrieves the date of the event.
- **getLocation() const:** Retrieves the location of the event.
- **getName() const:** Retrieves the name of the event.
- `addTeam` : add team or separate person which participate this event, also The function also iterates through all members of the team and adds their pointers to the event's participants list, ensuring no duplicates.
- `removeTeam : remove team which participate this event`
- **vector<Member> getParticipants() const*:** Retrieves the list of participants in the event.
- `getTeams`
- **Restrictions**
- **getid :** get the id
    - Events start and finish in the same date.
    - Only one event can be programming the same day

```
#include <vector>
#include <string>
#include "Member.h"

class Event {
private:
    std::string date;
    std::string location;
    std::string name;
    std::vector<Member*> participants;
   std::vector<Team*> teams;

public:
    Event(const std::string& date, const std::string& location, const
std::string& name);

    void reschedule(const std::string& new_date);
    void addParticipant(Member* participant);
    void removeParticipant(Member* participant);

    std::string getDate() const;
    std::string getLocation() const;
    std::string getName() const;
    std::vector<Member*> getParticipants() const;
   std::vector<Team*> getTeams() const;

    bool operator==(const Event& other) const;
    size_t getParticipantCount() const;
};
```

Addteam function:

```
// Add a team to the event
// Throws an exception if the team pointer is null or the team ID is invalid
void Event::addTeam(Team* team) {
    if (team == nullptr) {
        throw std::invalid_argument("Team pointer is null");  // Check for null
pointer
    }

    if (team->getId() <= 0) {
        throw std::invalid_argument("Team ID is invalid");  // Check for invalid
ID
    }

    // Check if the team already exists in the event by team ID
    auto it = std::find_if(teams.begin(), teams.end(), [team](Team* t) {
        return t->getId() == team->getId();
        });
    if (it != teams.end()) {
        throw std::invalid_argument("Team with this ID is already added to the
event");
    }

    teams.push_back(team);

    // Add all team members to the participants list
    for (auto member : team->getMembers()) {
        // Avoid duplicate participants
        if (member == nullptr) {
            throw std::runtime_error("Null member found in team");
        }
        if (std::find(participants.begin(), participants.end(), member) ==
participants.end()) {
            participants.push_back(member);
        }
    }
}
```

Event Class Tests

1. **Event Creation Test (Correct):** This test verifies that an Event object is created with the correct date and location.
2. **Reschedule Test (Correct):** Here, we check that the reschedule method updates an Event's date properly. After rescheduling, the test asserts that the date is updated as expected.
3. **Add Participant Test (Correct):** This test confirms that when a participant is added to an Event, the count of participants increases accordingly.
4. **Add Participant Test (Incorrect):** This test would aim to verify the Event class's handling of an invalid participant ID, such as a negative number. The test checks if the participant count does

not change after attempting to add an invalid participant. It's commented out in the test code, indicating that error handling for such cases might need to be implemented.

5. Test scheduling conflict for events

```
void testEvent() {
    bool allTestsPassed = true;

    // Event Creation Test (Correct)
    try {
        Event event("2024-05-01", "Stadium", "Football Match");
        assert(event.getDate() == "2024-05-01");
        assert(event.getLocation() == "Stadium");
        assert(event.getName() == "Football Match");
    }
    catch (...) {
        std::cout << "Event Creation Test failed" << std::endl;
        allTestsPassed = false;
    }

    // Reschedule Test (Correct)
    try {
        Event event("2024-05-01", "Stadium", "Football Match");
        event.reschedule("2024-06-01");
        assert(event.getDate() == "2024-06-01");
    }
    catch (...) {
        std::cout << "Reschedule Test failed" << std::endl;
        allTestsPassed = false;
    }

    // Add Participant Test (Correct)
    try {
        Event event("2024-05-01", "Stadium", "Football Match");
        Member* member1 = new Member("John", 30, "Athlete", 1);
        event.addParticipant(member1);
        assert(event.getParticipantCount() == 1);
        delete member1;
    }
    catch (...) {
        std::cout << "Add Participant Test failed" << std::endl;
        allTestsPassed = false;
    }

    // Add Participant Test (Incorrect)
    try {
        Event event("2024-05-01", "Stadium", "Football Match");
        try {
            event.addParticipant(nullptr); // This should fail or be handled
appropriately
            std::cerr << "Add Invalid Participant Test failed: no exception on
invalid participant" << std::endl;
            allTestsPassed = false;
        }
        catch (const std::invalid_argument& e) {
            // Expected exception, do nothing
        }
        assert(event.getParticipantCount() == 0);
    }
    catch (...) {
        std::cout << "Add Invalid Participant Test failed" << std::endl;
        allTestsPassed = false;
    }
```

```
    if (allTestsPassed) {
        std::cout << "testEvent passed" << std::endl;
    }
    else {
        std::cout << "testEvent failed" << std::endl;
    }
}


void testEventScheduleConflict() {
    try {
        Club club("Sports Club");

        Event* e1 = new Event("2024-04-19", "Stadium", "Football Match");
        Event* e2 = new Event("2024-04-20", "Gym", "Basketball Match");

        club.organizeEvent(e1);

        assert(club.hasScheduleConflict("2024-04-19"));
        assert(!club.hasScheduleConflict("2024-04-20"));

        std::cout << "testEventScheduleConflict passed" << std::endl;

        delete e1;
        delete e2;
    }
    catch (...) {
        std::cout << "testEventScheduleConflict failed" << std::endl;
    }
}
```

Relationships

1. **Club**
    - **The club is a container that holds all members, coaches, teams, and events.**
    - **The club can exist independently and does not depend on any other classes.**
2. **Member**
    - **Members can exist independently in the club without being associated with any specific coach or team.**
    - **Members can be added to teams and events as participants.**
    - **When a member is deleted, they are also removed from all associated teams and events.**
3. **Coach**
    - **Coaches can exist independently in the club without being associated with any specific team or member.**
    - **Coaches are usually related to members through teams; one coach can lead multiple teams.**
    - **When a coach is deleted, they are simply removed from the club without affecting members or teams.**

4. **Team**
    - **Teams can exist independently (without coach) .**
    - **Teams include multiple members and are led by a single coach.**
    - **When a team is deleted, it is removed from the club without affecting the coach or members.**
5. **Event**
    - **Events can exist independently in the club without being associated with any specific team or member.**
    - **Events include multiple members as participants.**
    - **When an event is deleted, it is removed from the club without affecting the members.**

General task: the purpose of this task is clearly show user how this "sport club" program working by using some standard function, so this task didn't consider more unexpected action(more completed task is after each class)

```
void printDivider(const std::string& title) {
    std::cout << "\n----------------------------------\n";
    std::cout << title << '\n';
    std::cout << "----------------------------------\n";
}

void testClubFunctionality() {
    // Create a club named "Elite Sports Club"
    printDivider("Creating Club: Elite Sports Club");
    Club club("Elite Sports Club");

    // Add members: dai, nick, and Bob
    printDivider("Adding Members: Jack, Kelly and Bob");
    Member* m1 = new Member("dai", 24, "Athlete", 1);
    Member* m2 = new Member("nick", 26, "Athlete", 2);
    Member* m3 = new Member("Bob", 30, "Athlete", 3);
    club.addMember(m1);
    club.addMember(m2);
    club.addMember(m3);

    // Print the current list of members
    std::cout << "Current Members:\n";
    for (const auto& member : club.getMembers()) {
        std::cout << "Name: " << member->getName() << ", Age: " << member-
>getAge() << ", Role: " << member->getRole() << ", ID: " << member->getId() <<
'\n';
    }

    // Add coaches: john (Tennis) and yang (Swimming)
    printDivider("Adding Coaches: Laura (Tennis) and Sam (Swimming)");
    Coach* c1 = new Coach("john", "Tennis", 1);
```

```
    Coach* c2 = new Coach("yang", "Swimming", 2);
    club.addCoach(c1);
    club.addCoach(c2);

    // Print the current list of coaches
    std::cout << "Current Coaches:\n";
    for (const auto& coach : club.getCoaches()) {
        std::cout << "Name: " << coach->getName() << ", Specialty: " << coach-
>getSpecialty() << ", ID: " << coach->getId() << '\n';
    }

    // Create and add teams: Tennis Team (coached by john) and Swimming Team
(coached by yang)
    printDivider("Creating and Adding Teams: Tennis Team and Swimming Team");
    Team* t1 = new Team("Tennis", c1, 1);
    Team* t2 = new Team("Swimming", c2, 2);
    t1->addMember(m1);
    t1->addMember(m2);
    t2->addMember(m3);
    club.addTeam(t1);
    club.addTeam(t2);

    // Print the current list of teams
    std::cout << "Current Teams:\n";
    for (const auto& team : club.getTeams()) {
        std::cout << "Sport Type: " << team->getSportType() << ", ID: " << team-
>getId() << '\n';
        std::cout << "Team Members:\n";
        for (const auto& member : team->getMembers()) {
            std::cout << "  Name: " << member->getName() << ", ID: " << member-
>getId() << '\n';
        }
    }

    // Organize events: Tennis Tournament and Swimming Competition
    printDivider("Organizing Events: Tennis Tournament and Swimming Competition");
    Event* e1 = new Event("2024-08-20", "City Arena", "Tennis Tournament");
    Event* e2 = new Event("2024-09-10", "Aquatic Center", "Swimming Competition");
    club.organizeEvent(e1);
    club.organizeEvent(e2);
    club.addMembersToEvent("Tennis Tournament", { m1, m2 });
    club.addMembersToEvent("Swimming Competition", { m3 });

    // Print the current list of events
    std::cout << "Current Events:\n";
    for (const auto& event : club.getEvents()) {
        std::cout << "Event Name: " << event->getName() << ", Date: " << event-
>getDate() << ", Location: " << event->getLocation() << '\n';
        std::cout << "Event Participants:\n";
        for (const auto& participant : event->getParticipants()) {
            std::cout << "  Name: " << participant->getName() << ", ID: " <<
participant->getId() << '\n';
        }
    }

    // Update coach john's specialty to Basketball
    printDivider("Updating Coach Specialty: Laura to Basketball");
    club.updateCoachSpecialty("Laura", "Basketball");

    // Print updated coach information
    std::cout << "Updated Coach:\n";
    for (const auto& coach : club.getCoaches()) {
        std::cout << "Name: " << coach->getName() << ", Specialty: " << coach-
```

```
>getSpecialty() << ", ID: " << coach->getId() << '\n';
    }

    // Check schedule conflict for 2024-08-20 and 2024-09-10
    printDivider("Checking Schedule Conflict for 2024-08-20 and 2024-09-10");
    bool conflict1 = club.hasScheduleConflict("2024-08-20");
    bool conflict2 = club.hasScheduleConflict("2024-09-10");
    std::cout << "Schedule Conflict on 2024-08-20: " << (conflict1 ? "Yes" : "No")
<< '\n';
    std::cout << "Schedule Conflict on 2024-09-10: " << (conflict2 ? "Yes" : "No")
<< '\n';

    // Find members by role
    printDivider("Finding Members by Role: Athlete");
    auto athletes = club.findMembersByRole("Athlete");
    for (const auto& member : athletes) {
        std::cout << "Name: " << member->getName() << ", ID: " << member->getId()
<< '\n';
    }

    // Find member by name
    printDivider("Finding Member by Name: Kelly");
    Member* foundMember = club.findMemberByName("Kelly");
    if (foundMember) {
        std::cout << "Found Member: " << foundMember->getName() << ", ID: " <<
foundMember->getId() << '\n';
    }
    else {
        std::cout << "Member not found\n";
    }

    // Remove member Jack (m1)
    printDivider("Removing Member: Jack");
    club.removeMember(m1);
    std::cout << "Current Members after removal:\n";
    for (const auto& member : club.getMembers()) {
        std::cout << "Name: " << member->getName() << ", ID: " << member->getId()
<< '\n';
    }

    // Remove coach Laura (c1)
    printDivider("Removing Coach: Laura");
    club.removeCoach(c1);
    std::cout << "Current Coaches after removal:\n";
    for (const auto& coach : club.getCoaches()) {
        std::cout << "Name: " << coach->getName() << ", ID: " << coach->getId()
<< '\n';
    }

    // Remove team Tennis Team (t1)
    printDivider("Removing Team: Tennis Team");
    club.removeTeam(t1);
    std::cout << "Current Teams after removal:\n";
    for (const auto& team : club.getTeams()) {
        std::cout << "Sport Type: " << team->getSportType() << ", ID: " << team-
>getId() << '\n';
    }

    // Cancel event Tennis Tournament (e1)
    printDivider("Canceling Event: Tennis Tournament");
    club.cancelEvent(e1);
    std::cout << "Current Events after cancellation:\n";
    for (const auto& event : club.getEvents()) {
```

```
        std::cout << "Event Name: " << event->getName() << ", Date: " << event-
>getDate() << ", Location: " << event->getLocation() << '\n';
    }

      delete m1;
      delete c1;
      delete e1;
      delete e2;
}
```

Memory map