Object Oriented Programming Solution
Question 01 -

a. A good programming practice is that the arguments of setter functions are constant, and getter functions themselves are constant functions. Why is that so? Please relate your answer to a real-world scenario.

In programming, keeping setter function arguments constant protects data, and making getter functions constant ensures they won't change data unexpectedly, like agreeing to specific rules in real life.

b. In object-oriented programming, why is it considered good practice to encapsulate data within classes and provide access to it through methods?

Encapsulating data within classes and accessing it through methods in object-oriented programming improves data security, modularity, and control over data manipulation, ensuring better code organization and reducing the risk of unintended side effects.

c. In object-oriented programming, methods of a class are generally made public so that they can be called from anywhere. However, can any method not be private? How can functions be made private? Please justify your answer with a real-world use case.

In object-oriented programming, methods are generally public for accessibility, but they can be made private for encapsulation. Private methods restrict access to within the class, ensuring data integrity and hiding implementation details. This is like how a chef keeps certain recipe steps private to maintain the uniqueness and quality of their dishes.

d. Why do we need a user-defined copy constructor if the compiler-defined copy constructor is already available?

A user-defined copy constructor may be needed to perform a deep copy or to handle specific resource management scenarios. While the compiler-provided copy constructor performs a shallow copy by default, a user-defined copy constructor allows customization to ensure proper copying of dynamically allocated memory or other resources.

e. What is the need to keep a static attribute within a class when it could be declared as a global variable?

Keeping a static attribute within a class provides better organization and encapsulation compared to declaring it as a global variable. By associating the attribute with a specific class, it clarifies its purpose and scope, making the code more understandable and maintainable. Additionally, it allows for better control over access rights, as the attribute can be made private or protected if needed, limiting direct modification from outside the class. This approach aligns with the principle of encapsulation in object-oriented programming, promoting better code structure and readability.

f. Which non-constant member function can be called by a constant object? Explain your answer.

constructor

Find the Error

```
#include <iostream>
using namespace std;
class ConstantValueContainer {
private:
    const int value;
    const int* const ptr;
public:
    ConstantValueContainer(int initialValue) : value(initialValue),
ptr(&value) {}
    void displayValue() const {
        cout << "Value through the constant pointer: " << *ptr
<<endl;
    }
 void modifyValue() { *ptr = 99; }
};

int main() {
    ConstantValueContainer container(42);
    container.modifyValue();
    container.displayValue();
}
```

Find the Error

```
class Complex
{
private:
double x,y;
static int z;
public:
Complex (double = 0.0);
static int doSomething() { z=2* y; return z; }
};
```

Question 02 - 10 marks (2+ 4*2)

```
#include <iostream>
#include <string>

using namespace std;

class Player {
private:
   string playerName;
```

```
    int ballScores[12];
public:
    Player(const string& name) : playerName(name) {
        // Initialize ballScores array with zeros
        for (int i = 0; i < 12; ++i) {
            ballScores[i] = 0;
        }
    }
    const string& getPlayerName() const {
        return playerName;
    }
    int* getBallScores() {
        return ballScores;
    }
    void updateScore(int ballNumber, int points) {
        ballScores[ballNumber - 1] = points;
    }
    int getTotalScore() const {
        int totalScore = 0;
        for (int i = 0; i < 12; ++i) {
            totalScore += ballScores[i];
        }
        return totalScore;
    }
    double getAverageScore() const {
        return getTotalScore() / 12.0;
    }

    void displayMatchScoreboard() {
        cout << playerName << "'s scores for each ball (in ascending order):\n";

        // Bubble sort for sorting scores
        int* scores = getBallScores();  // Use non-const pointer
        for (int i = 0; i < 12 - 1; ++i) {
            for (int j = 0; j < 12 - i - 1; ++j) {
                if (scores[j] > scores[j + 1]) {
                    // Swap scores
                    int temp = scores[j];
                    scores[j] = scores[j + 1];
                    scores[j + 1] = temp;
                }
            }
        }
```

```
        for (int i = 0; i < 12; ++i) {
            cout << "Ball " << (i + 1) << ": " << scores[i] << "\n";
        }

        cout << "Average score: " << getAverageScore() << "\n";
        cout << "Total score: " << getTotalScore() << "\n\n";
    }
};

class Game {
private:
    string gameName;

public:
    Game(const string& name) : gameName(name) {}

    const string& getGameName() const {
        return gameName;
    }

    void findWinner(const Player& player1, const Player& player2) const {
        int totalScorePlayer1 = player1.getTotalScore();
        int totalScorePlayer2 = player2.getTotalScore();
        cout << "Game Winner: ";
        if (totalScorePlayer1 > totalScorePlayer2) {
            cout << player1.getPlayerName() << " with a score of " << totalScorePlayer1 << "\n";
        } else if (totalScorePlayer2 > totalScorePlayer1) {
            cout << player2.getPlayerName() << " with a score of " << totalScorePlayer2 << "\n";
        } else {
            cout << "It's a tie! Both players have the same score of " << totalScorePlayer1 << "\n";
        }
    }

    void displayMatchScoreboard( Player& player1,  Player& player2)  {
        cout << "\nMatch Summary:\n";
        cout << player1.getPlayerName() << "'s scores:\n";
        player1.displayMatchScoreboard();
        cout << player2.getPlayerName() << "'s scores:\n";
        player2.displayMatchScoreboard();
    }
    int validateScore() {
    int score;
    cin >> score;
    score = (score >= 0 && score <= 6) ? score : 0;
```

```
    return score;
}

void playGame(Player& player,int playerNumber) {
    // Global function to simulate gameplay with an array of ball scores
    cout << "Playing " << this->getGameName() << " for Player " << playerNumber << ":\n";
    for (int ballNumber = 1; ballNumber <= 12; ++ballNumber) {
        cout << "Player " << playerNumber << ", enter the score for Ball " << ballNumber << "
(between 0 and 6): ";
        int ballScore = validateScore();  // Ensure the score is in the range of 0 to 6
        player.updateScore(ballNumber, ballScore);
    }
}
};



int main() {

    // Create instances of Player and Game
    Player player1("Player 1");
    Player player2("Player 2");
    Game cricketGame("Cricket");

    cricketGame.playGame(player1, 1);
    cricketGame.playGame(player2, 2);

    cricketGame.displayMatchScoreboard(player1, player2);
    cricketGame.findWinner(player1, player2);

    return 0;
}
```

Question 03

```
class Astronaut {
private:
    string name;
    string expertise;
    bool assignedToMission;

public:
    Astronaut() {}
```

```
    Astronaut(string name, string expertise) : name(name), expertise(expertise),
assignedToMission(false) {}
    ~Astronaut() {}

    string getName() const { return name; }
    string getExpertise() const { return expertise; }
    bool isAssigned() const { return assignedToMission; }
    void setAssigned(bool assigned) { assignedToMission = assigned; }
};

class Spacecraft {
private:
    string name;
    int capacity;
    bool missionReady;

public:
    Spacecraft(string name, int capacity) : name(name), capacity(capacity), missionReady(false)
{}

    string getName() const { return name; }
    int getCapacity() const { return capacity; }
    bool isMissionReady() const { return missionReady; }
    void setMissionReady(bool ready) { missionReady = ready; }
};

class Mission {
private:
    string name;
    int duration;
    string missionRequirements;
    string destination;
    Spacecraft* spacecraft;
    Astronaut* astronauts;
    int numAstronauts;

public:
    Mission(string name, int duration, string missionRequirements, string destination, Spacecraft*
spacecraft)
        : name(name), duration(duration), missionRequirements(missionRequirements),
destination(destination), spacecraft(spacecraft) {
        numAstronauts = 0;
        astronauts = new Astronaut[spacecraft->getCapacity()];
    }
```

```
    ~Mission() {
        delete[] astronauts;//needed because astronauts are dynamically allocated
    }

    string getName() const { return name; }
    int getDuration() const { return duration; }
    string getMissionRequirements() const { return missionRequirements; }
    string getDestination() const { return destination; }

    void addSpacecraft(Spacecraft* newSpacecraft) {
        spacecraft = newSpacecraft;
        spacecraft->setMissionReady(true);
    }

    void addAstronaut(Astronaut &newAstronaut) {
        if (numAstronauts < spacecraft->getCapacity() && !newAstronaut.isAssigned()) {
            for (int i = 0; i < numAstronauts; i++) {
                if (newAstronaut.getExpertise() == missionRequirements) {
                    astronauts[numAstronauts] = newAstronaut;
                    numAstronauts++;
                    newAstronaut.setAssigned(true);
                    break;
                }
            }
        }
    }
};
```