Assessed Exercises 1:

Exercise 1: Tennis

Part 1 – Writing the Tennis Class:

Created a class called TennisMatch, which stores the variables of a game and has a game loop function. It is interacted with via the console asking for the winning player of each round.

```
#include <iostream>
#include <string>
#include <iomanip> //setw

class TennisMatch{

    private:

    // Private Variables
    int score1= 0; int score2 = 0; int games1 = 0; int games2 = 0; //points and games
    int server = 1; //This keeps track of server (either 1 or 2)
    std::string player1, player2;
    bool running = true;
    int gameWinner = 0;
```

```
    // Functions
    void addScoreToWinner(){ //Takes and input and adds the points to the round winner.
        int lastWinner;
        std::cout << "\n";
        std::cout << "Input winner: ";
        std::cin >> lastWinner;
        std::cout << "\n" << std::endl;

        if (lastWinner==1){
            score1++;
        }
        if (lastWinner==2){
            score2++;
        }
    }
```

```cpp
    void announcement(){ // Umpire Annouces in "english" rather than scores.

        std::cout << " =-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=" << std::endl;
        if (gameWinner == 1){
            std::cout << "Umpire : game, " << player1 <<std::endl;
            gameWinner = 0;
        }
        if (gameWinner==2){
            std::cout << "Umpire : game, " << player2 <<std::endl;
            gameWinner = 0;
        }

        //Non server reliant annoucments
        if (score1==0 && score2==0){
            std::cout << "Umpire: " << "love-all" << std::endl;
        }
        if (score1==1 && score2==1){
            std::cout << "Umpire: " << "fifteen-all" << std::endl;
        }
        if (score1==2 && score2==2){
            std::cout << "Umpire: " << "thirty-all" << std::endl;
        }
        if ((score1>=3 && score2>=3) && (score1 == score2)){
            std::cout << "Umpire: " << "deuce" << std::endl;
        }

```

```cpp

        //Server reliant annoucements
        if (((score1 < 4 && score2 < 3) && (score1 != score2)) || ((score1 < 3 && score2 < 4) && (score1 != score2)) ){//Both less than 40
            if (server==1){
                std::cout << "Umpire: " << intToSpeach(score1, 1) << " - " << intToSpeach(score2, 2) << std::endl;
            }
            if (server==2){
                std::cout << "Umpire: " << intToSpeach(score2, 2) << " - " << intToSpeach(score1, 1) << std::endl;
            }
        }

        if (score1>=3 && score2>=3){
            if ((score1 == score2 + 1)){

                std::cout << "Umpire: " << "Advantage " << player1 << std::endl;
            }

            if ((score2 == score1 + 1)){

                std::cout << "Umpire: " << "Advantage " << player2 <<  std::endl;
            }
        }
        std::cout << std::endl;
    }
```

```cpp


    void scoreReading(){ //Prints out the scoreboard.
        std::cout << std::setw(19) << player1 << serving(1) << " | " << serving(2) << player2;  //Server
        std::cout << std::endl << std::setw(10) << "points| "  << std::setw(10) << intToScore(score1, 1) << " | " << intToScore(score2, 2);
        std::cout << std::endl << std::setw(10) << "games | "  << std::setw(10) << games1 << " | " << games2 << std::endl; //Games
    }


    std::string serving(int player){ //Is used to print the asterix next to the server.
        if (player == server){
            return "*";
        }
        return " ";
    }


```

```
    int scoreTranslation(int score, int player){ //The scoring logic. Returns a number so can't return "AD"

        if (score1<=3 && score2<=3){
            if (score==0){
                return 0;
            }
            if (score==1){
                return 15;
            }
            if (score==2){
                return 30;
            }
            if (score==3){
                return 40;
            }
        }

        if ((score1==3 && score2==2) || (score1==2 && score2==3)){
            if ((score1 > score2) && player==1){
                return 40;
            }
            if ((score2 > score1) && player==2){
                return 40;
            }
            return 30;
        }

        if (score1==3 && score2 == 3){
            return 40;
        }

        if (score1>=3 && score2>=3){
            if ((score1 == score2 + 1)){
                if (player==1){
                    return -10; // -10 = Advantage
                }
            }
            if ((score2 == score1 + 1)){
                if (player==2){
                    return -10; // -10 = Advantage
                }
            }
            return 40; // Watched a game of tennis and it goes back to 40 from "AD-40"
        }

        return -5;

    }
```

```cpp

std::string intToScore(int score, int player){ //Takes the result of the logic, and returns a string. (Implemented to get "AD")
    int value = scoreTranslation(score, player);

    switch (value){
        case 0: return "0";
        case 15: return "15";
        case 30: return "30";
        case 40: return "40";
        case -10: return "AD";
    }
    return "Error";
}


std::string intToSpeach(int score, int player){ //Returns what umpire should say.
    int value = scoreTranslation(score, player);

    switch (value){
        case 0: return "love";
        case 15: return "fifteen";
        case 30: return "thirty";
        case 40: return "fourty";
        case -10: return "advantage";
    }
    return "Error";
}
```

```cpp


int checkGameWon(){ //Checking if someone has won the game, uses the reset function.
    if (score1 >3 && score1 >= score2 + 2){
        games1++;
        reset();
        gameWinner = 1;
        return 1;
    }
    if (score2 >3 && score2 >= score1 + 2){
        games2++;
        reset();
        gameWinner = 2;
        return 2;
    }
    return 0;
}


void reset(){ //Resets the scores and swaps the player serving.
    score1 = 0;
    score2 = 0;

    if (server==1){
        server=2;
        return;
    }
    if (server==2){
        server=1;
```

```
        if (server==2){
            server=1;
            return;
        }

    }


    void checkMatchWon(){ //Checks if someone has 3 games. If they do, the game loop ends.
        if (games1>=3 || games2>=3){
            if (games1 > games2){
                std::cout << player1 << " Wins the Series!" << std::endl;

            }
            else{
                std::cout << player2 << " Wins the Series!" << std::endl;

            }
            running = false;
        }

    }
```

And now for the public functions:

```
    public:

    //Constructor
    TennisMatch(std::string _player1, std::string _player2){
        player1 = _player1;
        player2 = _player2;
    }

    //Functions
    void play(){ // function that runs everything when called.

        announcement();
        scoreReading();

        while (running){

            addScoreToWinner();
            checkGameWon();
            announcement();
            scoreReading();
            checkMatchWon();

            //announcement();

        }
    }
};
```

And the main where the class instance is set up with two names input by the software user.

The game loop function is then used.

```
int main(){

//Gets the inputs for the names of the people playing.
std::string string1; std::string string2;
std::cout << "Enter the names of the players: " << std::endl << "Player 1 : ";
std::cin >> string1;
std::cout << "Player 2 : ";
std::cin >> string2;
std::cout << std::endl;

//Initializes class
TennisMatch match(string1, string2);

//Starts game loop
match.play();

return 0;

}
```

Part 2 - Testing of the tennis loop:

For the testing, the program was run, and I entered in the names and the winners of each point.

Test 1:

Winning a game: Player 1 “George” wins all the points. Once he goes over 40, He gains a game, the points reset and the server swaps.

```
$ ./playTennis.exe
Enter the names of the players:
Player 1 : George
Player 2 : Alan

 =-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=
Umpire: love-all

             George* |  Alan
  points|          0 | 0
  games |          0 | 0

Input winner: 1


 =-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=
Umpire: fifteen - love

             George* |  Alan
  points|         15 | 0
  games |          0 | 0

Input winner: 1


 =-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=
Umpire: thirty - love

             George* |  Alan
  points|         30 | 0
  games |          0 | 0

Input winner: 1


 =-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=
Umpire: fourty - love

             George* |  Alan
  points|         40 | 0
  games |          0 | 0

Input winner: 1


 =-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=
Umpire : game, George
Umpire: love-all

             George  | *Alan
  points|          0 | 0
  games |          1 | 0

Input winner: |
```

Test 2: Checking the announcements. Such as deuce at 40-40 and advantage.

Here it shows that when someone scores into advantage, the Umpire calls “Advantage ‘player’”.

Then if the other player scores, the advantage is gone, and it returns to deuce as it does in professional tennis games.

It also shows player 2 getting a game, and the server switching back to the other player.

```
Input winner: 1


 =-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=
Umpire: love - fifteen

             George   | *Alan
  points|         15 | 0
  games |          1 | 0

Input winner: 2


 =-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=
Umpire: fifteen-all

             George   | *Alan
  points|         15 | 15
  games |          1 | 0

Input winner: 1


 =-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=
Umpire: fifteen - thirty

             George   | *Alan
  points|         30 | 15
  games |          1 | 0

Input winner: 2


 =-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=
Umpire: thirty-all

             George   | *Alan
  points|         30 | 30
  games |          1 | 0

Input winner: 1


 =-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=
Umpire: thirty - fourty

             George   | *Alan
  points|         40 | 30
  games |          1 | 0

Input winner: 2


 =-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=
Umpire: deuce

             George   | *Alan
  points|         40 | 40
  games |          1 | 0
```

```
 =-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=
Umpire: deuce

             George   | *Alan
  points|         40 | 40
  games |          1 | 0

Input winner: 1


 =-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=
Umpire: Advantage George

             George   | *Alan
  points|         AD | 40
  games |          1 | 0

Input winner: 2


 =-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=
Umpire: deuce

             George   | *Alan
  points|         40 | 40
  games |          1 | 0

Input winner: 2


 =-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=
Umpire: Advantage Alan

             George   | *Alan
  points|         40 | AD
  games |          1 | 0

Input winner: 1


 =-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=
Umpire: deuce

             George   | *Alan
  points|         40 | 40
  games |          1 | 0

Input winner: 2


 =-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=
Umpire: Advantage Alan

             George   | *Alan
  points|         40 | AD
  games |          1 | 0

Input winner: 2


 =-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=
Umpire : game, Alan
Umpire: love-all

             George*  |  Alan
  points|          0 | 0
  games |          1 | 1
```

Test 3: Checking the end of the game loop

The winner is the first player to get to three games:

Player 1 “Georges” gets to three games. The umpire says “game, George” and then the program prints “George wins the series” and then it ends.

```
=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=
Umpire: love - fourty

             George   | *Alan
  points|          40 | 0
  games |           2 | 1

Input winner: 1


 =-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=
Umpire : game, George
Umpire: love-all

             George* |  Alan
  points|           0 | 0
  games |           3 | 1
George Wins the Series!
```

Exercise 2: Rational number class

Part 1 – Writing the class:

Made a class which holds a numerator and a denominator of a fraction. Overloaded lots of operators so that this class can be easily used in other methods, such as a regular Falsi method.

```cpp
#include <iostream>

class Fraction{

    private:

    // Variables
    long long numerator; long long denominator; //Long ints to get larger range of fractions.

    public:

    //Constructor
    constexpr Fraction(long long _numerator,long long _denominator):
    numerator(_numerator),
    denominator(_denominator){
        if (_denominator==0){//Check to make sure not dividing by zero.
            throw std::logic_error("Divided by 0. You cannot have a denominator of 0.");
        }
        simplify(); // Simplify the fraction on initialisation.
    };
```

Constructor checks whether the denominator is 0. If it is, an error is thrown.

The fractions are simplified in the initialization so that 2/4 == ½.

```cpp
    //Functions

    friend std::ostream& operator<<(std::ostream& os, const Fraction& fraction){ //Print to console (cout)
        std::cout << fraction.numerator << "/" << fraction.denominator << std::endl;
        return os;
    }

    long long gcf(const long long &a,const long long &b){ //Get common factor
        long long c = a % b;
        if (c == 0){
            return b;
        }

    return gcf(b,c);
    }

    void simplify(){//Simplifies to fraction as much as possible
        long long bcf = gcf(numerator, denominator);

        numerator /= bcf;
        denominator /= bcf;

        if (denominator < 0){
            numerator *= -1;
            denominator *= -1;
        }
    }
```

Now for all the operators:

```cpp
    Fraction operator +(Fraction const &other){ //Addition
        long long newNum = numerator*other.denominator + other.numerator*denominator;
        long long newDen = denominator*other.denominator;

        Fraction sum(newNum,newDen);

        return sum;
    }

    Fraction operator -(Fraction const &other){ // Subtraction
        long long newNum = numerator*other.denominator - other.numerator*denominator;
        long long newDen = denominator*other.denominator;

        Fraction sum(newNum,newDen);

        return sum;
    }

    Fraction operator *(Fraction const &other){ // Mulitplication
        long long newNum = numerator*other.numerator;
        long long newDen = denominator*other.denominator;

        Fraction product(newNum,newDen);

        return product;
    }
```

```
    Fraction operator /(Fraction const &other){ // Division
        long long newNum = numerator*other.denominator;
        long long newDen = denominator*other.numerator;

        Fraction product(newNum,newDen);

        return product;
    }

    Fraction& operator +=(Fraction const &other){
        *this = *this + other;
        return *this;
    }

    Fraction& operator -=(Fraction const &other){
        *this = *this - other;
        return *this;
    }

    Fraction& operator /=(Fraction const &other){
        *this = *this / other;
        return *this;
    }

    Fraction& operator *=(Fraction const &other){
        *this = *this * other;
        return *this;
    }
```

```
    constexpr Fraction& operator++(){ //Prefix ++
        numerator = numerator + denominator;

        return *this;
    }

    constexpr Fraction operator++(int){ //Postfix ++
        Fraction holder = *this;
        ++(*this);
        return holder;
    }

    constexpr Fraction& operator--(){ //Prefix --
        numerator = numerator - denominator;
        return *this;
    }

    constexpr Fraction operator--(int){ //Postfix ++
        Fraction holder = *this;
        --(*this);
        return holder;
    }
```

```
    bool operator==(const Fraction &other){
        if (numerator == other.numerator && denominator == other.denominator){
            return true;
        }
        return false;
    }

    bool operator!=(const Fraction &other){

        if (numerator != other.numerator || denominator != other.denominator){
            return true;
        }
        return false;
    }

    bool operator>(const Fraction &other){
        double forigin = (long double)numerator/(long double)denominator;
        double fother  = (long double)other.numerator/(long double)other.denominator;

        if (forigin > fother){
            return true;
        }
        return false;

    }
```

```
    bool operator<(const Fraction &other){
        double forigin = (long double)numerator/(long double)denominator;
        double fother  = (long double)other.numerator/(long double)other.denominator;

        if (forigin < fother){
            return true;
        }
        return false;
    }

    bool operator>=(const Fraction &other){
        double forigin = (long double)numerator/(long double)denominator;
        double fother  = (long double)other.numerator/(long double)other.denominator;

        if (forigin >= fother){
            return true;
        }
        return false;
    }

    bool operator<=(const Fraction &other){
        double forigin = (long double)numerator/(long double)denominator;
        double fother  = (long double)other.numerator/(long double)other.denominator;

        if (forigin <= fother){
            return true;
        }
        return false;
    }
```

Some other useful functions:

```
    Fraction abs(){//Absolute of the fraction
        simplify();
        long long store = numerator;
        if (store < 0){
            store *=-1;
        }
        Fraction absolute(store, denominator);
        return absolute;

    };

    constexpr Fraction pow(int x){ //To the power of
        Fraction result(1,1);

        for (unsigned i = 0;  i < x;  ++i){
            result *= *this;
        }

        return result;
    }
};
```

The class is now finished. All the methods are defined inside the class due to personal preference.

Testing the class:

```
int main(){
    Fraction first(7,15);
    Fraction second(6,29);

    std::cout << "\n+ : " << first + second << std::endl; //expected : 293/435
    std::cout << "- : " << first - second << std::endl; //expected : 113/435
    std::cout << "* : " << first * second << std::endl; //expected : 14/145
    std::cout << "/ : " << first / second << std::endl; //expected : 203/90

    first += Fraction(1,1);
    std::cout << "+= " << first << std::endl; //expected : 22/15

    first++;
    std::cout << "++ " <<first; //expected : 37/15

        return 0;
}
```

Provides an overview of the basic operators and gives the output:

```
$ ./a.exe

+ : 293/435

- : 113/435

* : 14/145

/ : 203/90

+= : 22/15

++ : 37/15
```

This time with a negative number: Note that it does not matter if the minus sign is included in the numerator or denominator as the simplify function will correct it.

```cpp
int main(){
    Fraction first(-7,15);
    Fraction second(6,29);

    std::cout << "\n+ : " << first + second << std::endl; //expected : -113/435
    std::cout << "- : " << first - second << std::endl; //expected : -293/435
    std::cout << "* : " << first * second << std::endl; //expected : -14/145
    std::cout << "/ : " << first / second << std::endl; //expected : -203/90

    first += Fraction(1,1);
    std::cout << "+= : " << first << std::endl; //expected : 8/15

    first++;
    std::cout << "++ : " <<first; //expected : 23/15

        return 0;
}
```

Testing the pow and abs functions

```cpp
int main(){
    Fraction first(-7,15);
    Fraction second(6,5);

    std::cout << first << "Squared is : " << first.pow(2) << std::endl;
    std::cout << second << "Cubed is : " << second.pow(3) << std::endl;

    std::cout << first << "The absolute is : " << first.abs() << std::endl;


    return 0;
}
```

Returns:

```
-7/15
Squared is : 49/225

6/5
Cubed is : 216/125

-7/15
The absolute is : 7/15
```

Now using a few to combine into something slightly more complicated:

```cpp
int main(){
    Fraction half(1,2);
    Fraction six(6,1);
    Fraction third(1,3);
    Fraction nine(9,1);
    std::cout << "((6 * 0.5 )/(1/3))*9. Would expect 81:" << std::endl;
    std::cout << "we get: " << (six * half)/third * nine << std::endl;


    return 0;
}
```

```
((6 * 0.5 )/(1/3))*9. Would expect 81:
we get: 81/1
```

Part 2 – Implementing Regula Falsi:

Now that the class has the mathematical operators, we can start to use it to build more complex programs.

Using the class to find a root using the regular Falsi method.

Here a function for the Regular Falsi is defined as well as the equation x^2 – 2 which we want to find the route for.

```
Fraction findRootFalsi(Fraction (*f)(Fraction),Fraction x1, Fraction x2){
    Fraction tolerance(1ll,100000000ll); //Tolerance of 10^-8
    Fraction zero(0ll,1ll);              // Just a 0 fraction 0/1

    Fraction middle = (x1*(*f)(x2)  -  x2*(*f)(x1)) / ((*f)(x2) - (*f)(x1));

    while( ((*f)(middle)).abs() > tolerance){// While the absolute value at middle is greater than the tolerance away from 0.

        if ( (*f)(middle) <= zero){ //If the value at middle is below 0. Set x1 to middle.
            x1 = middle;
        }
        else{                        //If the value at middle is above 0. Set x2 to middle.
            x2 = middle;
        }
        middle = (x1 * (*f)(x2)  -  x2*(*f)(x1)) / ((*f)(x2) - (*f)(x1));
        std::cout << (middle) << std::endl;
        }
    //}
    return middle;
    }
```

The regular falsi can be used to find a root of an equation (provided it has a root):

The assessment asked for the root of the function x^2 – 2, between 0 and 2.

Running the main function:

```
int main(){

    Fraction root = findRootFalsi(myQuadratic, Fraction(0,1), Fraction(2,1));
    std::cout << "\n Root of x^2 -2 : " << root << std::endl;

    return 0;
}
```

Where myFunction is :

```
Fraction myQuadratic(Fraction x){ // Simple quadratic to show off regular falsi y = x^2 - 2
    return (x * x) -Fraction(2,1);
}
```

Returns:

```
4/3

7/5

24/17

41/29

140/99

239/169

816/577

1393/985

4756/3363

8119/5741

27720/19601


Root of x^2 -2 : 27720/19601
```

Which when put into a calculator gives the value: 1.414213561.

And the actual root is at 1.414213562....

The answer is extremely close as expected with the low tolerance.

Part 3 – Arising problems and limitations to a rational number class:

One limitation is the amount of memory each object consumes. If you want a precise number, the numerator and denominator need to be sufficiently big enough to accommodate for that accuracy. An extremely small number requires a huge denominator, as well as a numerator. If you are using these fraction classes on a grand scale, it would be more memory efficient to use doubles or long doubles. Limiting the effectiveness of using this class for high performance programs.

Overflows could become a problem since some numbers require exceptionally large numerators and/or denominators. Therefore, when using this rational number class to do some arithmetic, overflow errors can pop up due to the size of the answer's object's value holders being too small to hold the new number. For example, 0.124349729340 requires large values for its numerator and denominator. Multiplying this number by some other number, say, 22.13457903 which also includes large values for the numerator and denominator, then a completely wrong answer could arise due to an overflow error resulting from the extremely large values that do not fit into the memory allocated for the numerator and denominator in the class. This limits the usefulness of the class for uses where high accuracy is required.

An arising problem would be the usefulness of displaying the fractions. When the number gets exceptionally large, printing out the numerator and the denominator becomes difficult to read, and to get an understanding of the actual number it represents requires the use of a calculator. For example, getting an output of 134797495902/2425793992 is difficult for a human to comprehend, whereas if it is displayed in a different form such as a double, it is simply 55.56840208.