```cpp
#include <algorithm>
#include <numeric>          // To use GCD
#include <cmath>
#include <iostream>
#include "Assignment6.h"

using namespace std;

////////////////////////////////////////////////////////////
//            METHODS/FUNCTIONS DEFINITIONS               //
////////////////////////////////////////////////////////////

// Default constructor - To set an object to a known state
fraction::fraction(void) {
    n = 0;
    d = 1;
}

// Basically another constructor...
void fraction::setFraction(int n, int d) {
    this->n = n;
    this->d = d;
}


void fraction::setFraction(fraction f) {
    this->n = f.n;
    this->d = f.d;
}


// From here on, notice that it makes no sense to simplify if
// either the numerator or the denominator are equal to 1 or if
// the numerator is equal to 0.

//**********************************************************

void fraction::add(fraction f) {
    n = (f.d * n + f.n * d);
    d *= f.d;
    if (n != 0 && n != 1 && d != 1)
        simplify();
}


//************************************************************
void fraction::mult(fraction f) {
    n *= f.n;
    d *= f.d;
    if (n != 0 && n != 1 && d != 1)
        simplify();
}
```

```cpp

//*************************************************************
void fraction::div(fraction f) {
    n = n * f.d;
    d = d * f.n;
    if (n != 0 && n != 1 && d != 1)
        simplify();
}


//*************************************************************
void fraction::display(void) {
    if (n == 1 && d == 1)
        cout << 1 << endl;
    else if (n == 0)
        cout << 0 << endl;
    else if (d == 1)
        cout << n << endl;
    else
        cout << n << " / " << d << endl;
}


//*************************************************************
// NB: To use std::gcd be sure that the IDE is set to C++17
// for the project. Because of this I originally wrote my own
// functions (that I left below).

void fraction::simplify() {
    int m;
    // Simple cases first because computation is elementary
    if (n == d) {
        n = 1;
        d = 1;
    }
    else if ((d % n == 0) && (n < d)) {
        d /= n;
        n = 1;
    }
    else if ((n % d == 0) && (n > d)) {
        n /= d;
        d = 1;
    }
    else {
        m = (n > d) ? std::gcd(n, d) : std::gcd(d, n);

        // m = naiveGCD()
        // m = (n > d) ? euclidianGCD(n, d) : euclidianGCD(d, n);
        // m = (n > d) ? binaryGCD(n, d) : binaryGCD(d, n);
        n /= m;
        d /= m;
    }
}
```

```cpp
//*************************************************************
// Naive solution

int fraction::naiveGCD() {
    int gcd;
    for (int i = 1; i <= n && i <= d; i++)
        if (n % i == 0 && d % i == 0)
            gcd = i;
    return gcd;
}


// Euclidean Algorithm for GCD - Recursive
// Should be very efficient for small numbers though I'm not sure
// how small is "small"...

int fraction::euclidianGCD(int a, int b) {
    if (a == 0) // probably unnecessary
        return b;
    else if (b == 0)
        return a;
    else {
        return euclidianGCD(b, a % b);
    }
}


// Binary GCD - Recursive
// It should be even more efficient than Euclidian.

int fraction::binaryGCD(int a, int b) {
    if (a == 0)             // probably unnecessary
        return b;
    if (b == 0)
        return a;
    // a is even
    if (a % 2 == 0)

        if (b % 2 == 0)
            // both are even
            return 2 * binaryGCD(a / 2, b / 2);
        else
            // a is even and b is odd
            return binaryGCD(a / 2, b);
    // a is odd
    else
        // a is odd and b is even
        if (b % 2 == 0)
            return binaryGCD(a, b / 2);

        else
            // both are odd
            return binaryGCD(abs(a - b) / 2, b);

    // in case of mistakes... and to make some compilers happy
    return 0;
}
```