```cpp
#include <iostream>
#include <sstream>
#include <string>
#include <iomanip>
#include "Assignment6.h"


// Uncomment this to use in_stream.ignore
// auto max_size = std::numeric_limits<std::streamsize>::max();


void calculate(std::stringstream & in_stream, fraction & result) {
    char c;
    int numerator, denominator;
    fraction f;
    std::string ops;

    while (!in_stream.eof()) {
        // Read numerator
        in_stream >> numerator;

        // Drop/skip the fraction sign '/'
        // in_stream.ignore(max_size, '/');
        in_stream >> c;

        // Read denominator
        in_stream >> denominator;

        // Create the fraction
        f.setFraction(numerator, denominator);

        // Check if there's an operator saved
        if (ops == "+")
            result.add(f);
        else if (ops == "*")
            result.mult(f);
        else if (ops == "div")
            result.div(f);
        else if (ops.empty())
            result.setFraction(f);
        else {
            std::cout << "error" << std::endl;
            return;
        }

        // Ignore all white spaces until the next "non-white"
        // character that can be peeked
        // NB In some compilers this may not be necessary
        in_stream >> std::ws;

        // Read operator
        if (!in_stream.eof())
            if (in_stream.peek() == '+' || in_stream.peek() == '*')
                in_stream >> std::setw(1) >> ops;
        if (in_stream.peek() == 'd')
            in_stream >> std::setw(3) >> ops;
    }
}
```

```cpp




/////////////////////////////////////////////////////////////////////




// How it works:
// It reads a fraction and checks if there is an operator saved.
// If there's none, it means that the first operand was read and
// the result is just set to the fraction.
// If there's an operator saved, the fraction is updated by the
// corrisponding operation (this in-place behaviour is required
// by the assignment).
//
// Example:
//
//  2 / 3 + 3 / 7
//
// 1. Read 2/3 (loop 1)
// 2. There's no operator saved: result = 2/3 (loop 1)
// 3. Save '+' (loop 1)
// 4. Read 3/7 (loop 2)
// 5. There's an operator '+' saved (loop 2)
// 6. result = 2/3 + 3/7 = 23/21

//////////////////////////////////////////////////////////////////////////

// Comments:

// * One way to drop '/' is to import it into a character and then
// forget about it.
// Though reading '/' and not using it is not very elegant, the
// mechanism to skip data in a stringstream is overly complicated
// for just one single character. Also it requires to create an instance
// of a class that is not used for anything else and might take more space
// than a simple char.
// I'm not sure what happens behind the scenes and what is actually
// more efficient...

// Because of the different lenght of operators that have to be checked,
// the streamm needs to be peeked to know how many characters to read into 'ops'
```

```
///////////////////////////////////////////////////////////////////////////
// The same function can operate recursively.
// It works in a way that's easier to understand:
// - Read a fraction
// - Read an operator
// - Call the function recursively in the context of the operation specified
////////////////////////////////////////////////////////////////////////////


fraction calculateRecursive(std::stringstream & in_stream) {
    fraction f;
    std::string ops;
    char c;
    int numerator, denominator;

    if (!in_stream.eof()) {
        in_stream >> numerator;
        in_stream >> c;
        in_stream >> denominator;

        f.setFraction(numerator, denominator);

        in_stream >> std::ws;

        if (!in_stream.eof())
            if (in_stream.peek() == '+' || in_stream.peek() == '*')
                in_stream >> std::setw(1) >> ops;
            if (in_stream.peek() == 'd')
                in_stream >> std::setw(3) >> ops;

        if (ops == "+")
            f.add(calculateRecursive(in_stream));
        else if (ops == "*")
            f.mult(calculateRecursive(in_stream));
        else if (ops == "div")
            f.div(calculateRecursive(in_stream));
    }
    return f;
}



///////////////////////////////////////////////////////////////////////////

// I don't like either solution... They're a bit too "twisted".
```