CISS245: Advanced Programming
Assignment a06

Objectives: The purpose of this assignment is to build a simple library for a struct.

1. Declare a struct variable.
2. Access member variables of a struct variable.
3. Write function/operator with struct parameters.
4. Write function/operator with struct return value.

For CISS245, I also emphasize rigorous software testing. Much of the testing strategies that you will learn in this class (although in a small scale) is actually being practiced in the real software engineering world. Specifically, for this and future assignments I will emphasize the testing of the smallest units of a software such as functions. These are called **unit tests**.

Structure Variables

A structure variable is just a variable that contains variables (refer to your notes). Run this:

```
#include <iostream>
#include <iomanip>

struct Time
{
    int hour;
    int min;
    int sec;
}

int main()
{
    Time t0;
    t0.hour = 5;
    t0.min = 18;
    t0.sec = 0;

    std::cout << std::setw(2) << std::setfill('0') << t0.hour
              << ':'
              << std::setw(2) << std::setfill('0') << t0.min
              << ':'
              << std::setw(2) << std::setfill('0') << t0.sec
              << std::endl;

    return 0;
}
```

In this case, I want to work with time which is represented by hour, minute, and second. However, I frequently want to view hour, minute, and second not as three things but one. So I create this new type:

```
struct Time
{
    int hour;
    int min;
    int sec;
}
```

I can then create a `Time` (structure) variable like this:

```
Time t0;
```

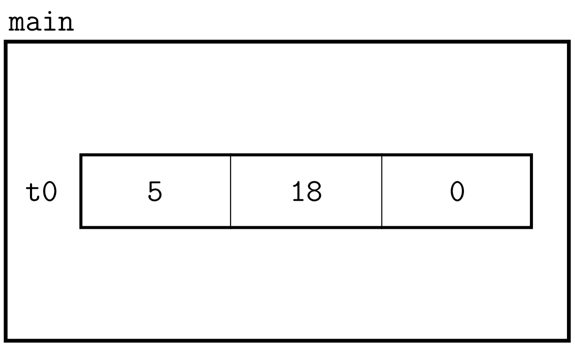
And now `t0` contains three variables. `t0` has:

- `t0.hour` which is an `int` variable.
- `t0.min` which is an `int` variable.
- `t0.sec` which is an `int` variable.

The variables inside a structure variable are usually called **member variables**.

Memory model

Here's the memory model of `main()` for the variable `t0` in the above example:



(Like I said, `t0` is just a variable containing variables.)

Here's another example of a structure type:

```
struct Student
{
    int id;
    double gpa;
}
```

In this case, a `Student` variable contains 2 variables. For instance, after this declaration:

```
Student jdoe;
```

the variable `jdoe` contains:

- `jdoe.id`, an `int` variable and
- `jdoe.gpa`, a `double` variable.

Note that the variables in a structure can have different types. In the `Student` case, there is one `int` variable and one `double` variable.

Note also that to go into a variable inside a structure variable, you use the dot. For instance, to go to the hour of `t0`, you use:

```
t0.hour
```

The variables inside a structure variable are just like any regular variable so you do know how to work with them (input, output, operators etc.) For instance, back in the first `Time` example, we assign values to the variable `t0`:

```
t0.hour = 5;
t0.min = 18;
t0.sec = 0;
```

We also read the values in `t0` and print them:

```
std::cout << std::setw(2) << std::setfill('0') << t0.hour
          << ':'
          << std::setw(2) << std::setfill('0') << t0.min
          << ':'
          << std::setw(2) << std::setfill('0') << t0.sec
          << std::endl;
```

(By the way, although a structure variable is sort of like an array, you see from the above that there are differences. For instance, an array can only contain values of the same type. You can't have an array of 2 integers and 3 doubles. An array is an `int` array or a `double` array. You can't have it both ways! To go into an array, you have the bracket `[]` with an index value. You get into a structure using the dot and a name for the variable inside the structure that you're trying to access.)

Initialization

Instead of doing this:

```
Time t0;
t0.hour = 5;
t0.min = 18;
t0.sec = 0;
```

to initialize structure variables, you can use the same notation as in array initialization:

```
Time t0 = {5, 18, 0};
```

Functions: struct parameters

Just like variables of basic types, it's not surprising that you can pass structure variables into a function. Run this:

```
#include <iostream>
#include <iomanip>

struct Time
{
    int hour;
    int min;
    int sec;
};

void print(Time t)
{
    std::cout << std::setw(2) << std::setfill('0') << t.hour
              << ':'
              << std::setw(2) << std::setfill('0') << t.min
              << ':'
              << std::setw(2) << std::setfill('0') << t.sec
              << std::endl;
}

int main()
{
    Time t0 = {5, 18, 0};
    print(t0);

    return 0;
}
```

Structure variables in function calls are by default pass–by–value. This means that the variable inside the function cannot change the variable in the calling function. Try this:

```
...

void inc(Time t)
{
    t.sec++;
}

...

int main()
{
    ...
    inc(t0);
    print(t0); // t0.sec is the same!!!

    return 0;
}
```

The variable `t0` is not changed when you return from the function call to `inc()`. So, **if you do want to change `t0`**, you have to force a pass–by–reference like this:

```
...

void inc(Time & t)
{
    t.sec++;
}

...

int main()
{
    ...
    inc(t0);
    print(t0); // t0.sec has changed!!!

    return 0;
}
```

So, in terms of parameter–passing, structure variables are like `int` and `double` parameters which are also pass–by–value.

Function: Reference–to–struct parameters

For performance reasons (i.e., speed), we force all **structure variables to be passed by reference**. Make this change and run your program again:

```
void print(Time & t)
{
    std::cout << std::setw(2) << std::setfill('0') << t.hour
              << ':'
              << std::setw(2) << std::setfill('0') << t.min
              << ':'
              << std::setw(2) << std::setfill('0') << t.sec
              << std::endl;
}
```

This avoids creating memory for `t`; `t` simply references the variable in the calling function.

If a `struct` parameter is a reference, the function can change the variable in the calling function:

```
void print(Time & t)
{
    std::cout << std::setw(2) << std::setfill('0') << t.hour
              << ':'
              << std::setw(2) << std::setfill('0') << t.min
              << ':'
              << std::setw(2) << std::setfill('0') << t.sec
              << std::endl;
    // This will change the Time variable that t references!!!
    t.sec = 0;
}
```

This means that the `print()` function now has the ability to change the variable in the calling function. To avoid accidentally changing this variable, we force the parameter `t` to be constant:

```
void print(const Time & t)
{
    std::cout << std::setw(2) << std::setfill('0') << t.hour
              << ':'
              << std::setw(2) << std::setfill('0') << t.min
              << ':'
              << std::setw(2) << std::setfill('0') << t.sec
              << std::endl;
}
```

So, let me repeat myself:

All stucture variables must be passed by reference (it's also possible to pass the address of the `struct` variable to a pointer receiving the address – see later.) If a function wants to change the variable in the calling function, there's nothing else to do. If a function should not change the variable in the calling function, make the parameter `const` as well.

Functions: struct return value

You can return a structure value:

```
...

Time addOneHour(const Time & t)
{
    Time newtime = t;
    newtime.hour++;

    return newtime;
}

int main()
{
    ...

    Time t1 = addOneHour(t0);
    print(t1);

    return 0;
}
```

(So, structure variables are also different from arrays because you can return them. Recall that you cannot return arrays.)

Functions: pointer–to–struct parameters

As mentioned earlier, it's also possible to pass the address of a `struct` variable. Try this version:

```
void print(const Time * t)
{
    std::cout << std::setw(2) << std::setfill('0') << t->hour
              << ':'
              << std::setw(2) << std::setfill('0') << t->min
              << ':'
              << std::setw(2) << std::setfill('0') << t->sec
              << std::endl;
}

...

int main()
{
    Time t0;
    t0.hour = 5;
    t0.min = 18;
    t0.sec = 0;
    print(&t0);

    return 0;
}
```

where `t->hour` is really just `(*t).hour`.

The corresponding `addOneHour()` function looks like this:

```
Time addOneHour(const Time * t)
{
    Time newtime = *t;
    newtime.hour++;

    return newtime;
}

int main()
{
    ...
    Time t1 = addOneHour(&t0);

    return 0;
}
```

Let me summarize:

- If you want a function to work with a `struct` value, instead of pass–by–value, you should use either pass–by–reference or pass the address, i.e., the receiving parameter in the function must be a reference or a pointer.

- If the parameter in the function should not change the `struct` value from the calling function, then the parameter in the function must be a constant reference or a pointer to a `const` value.

Now for the assignment

The goal is to build a simple library for computing fractions and test it rigorously.

To make it easy for you, I have written it in parts.

The questions build upon each other. This means that subsequent questions are going to rely significantly on previous questions. Once you are done with a question, you make a copy of that folder for the next question and continue work on the next folder. For instance if you are done with question 3, you make a copy of the folder for question 3 and rename it appropriately for question 4 and continue work on the folder for question 4. Etc.

The Given Code Base

Note that C++ does not natively support fractions. (Doubles are not fractions.) In this assignment we will be developing useful functions and operators to support the use of fractions. I'll give you the structure definition:

```
struct Fraction
{
    int n; // numerator
    int d; // denominator
};
```

This means of course that if you want to model the mathematical fraction

$$x = \frac{3}{4}$$

you would do this in your program:

```
Fraction x;
x.n = 3;
x.d = 4;
```

or better:

```
Fraction x = {3, 4};
```

You will need to write three files:

- `test_fraction.cpp`: This contains a program to test the `Fraction struct` and its supporting functions and operators.
- `Fraction.h`: This is the header function containing the `Fraction struct` and its prototypes.
- `Fraction.cpp`: This file contains the definition of the prototypes in `Fraction.h`.

The following are the skeleton files (remember that skeleton files are incomplete and might contain errors):

```
// Author:
// Date  :
// File  : test_fraction.cpp

#include <iostream>
#include "Fraction.h"


void test_print()
```

```
{
    int n = 0, d = 0;
    std::cin >> n >> d;
    Fraction f = {n, d};
    std::cout << f << std::endl;
}


int main()
{
    int option;
    std::cin >> option;

    switch (option)
    {
        case 1:
            test_print();
            break;
    }

    return 0;
}
```

(Note: the tester uses option 1 to test the print feature.)

```
// Author:
// Date  :
// File  : Fraction.h

#ifndef FRACTION_H
#define FRACTION_H

#include <iostream>

struct Fraction
{
    int n; // numerator
    int d; // denominator
};


std::ostream & operator<<(std::ostream &, const Fraction &);
```

```
#endif
```

```
// Author:
// Date  :
// File  : Fraction.cpp

#include <iostream>
#include "Fraction.h"


std::ostream & operator<<(std::ostream & cout, const Fraction & r)
{
    cout << r.n << '/' << r.d;
    return cout;
}
```

You should compile (correcting any errors if necessary) and run the program to make sure that it works before continuing. Note that in `Fraction.cpp` the `operator<<` is defined. Right now, the only thing you need to know is that in the code for `operator<<`, if you want to print `r.n` (say), call

```
cout << r.n
```

instead of

```
std::cout << r.n
```

i.e.:

```
// Author:
// Date  :
// File  : Fraction.cpp

#include <iostream>
#include "Fraction.h"


std::ostream & operator<<(std::ostream & cout, const Fraction & r)
{
    cout << r.n << '/' << r.d;
    return cout;
}
```

This is how you should print inside this "function" (technically speaking this is an operator, not a function). Also, do not modify the prototype of this "function" nor remove the last statement in the body:

```
std::ostream & operator<<(std::ostream & cout, const Fraction & r)
{
    ...
    return cout;
}
```

In more detail, in `main()`, suppose you have `Time` variable called `r`, if you execute

```
std::cout << r;
```

C++ actually executes

```
operator<< (std::cout, r);
```

i.e., it calls the "function" (or rather the operator) `operator<<` so that in this "function" the parameter `cout` references `std::cout` and the reference variable `r` references the `r` in `main()`.

Compiling and Automating Test Inputs

The following instructions are for those using Fedora.

Be neat! Have a directory just for this program and do not have your source files cluttered with other things! To compile all your source files you do

```
g++ *.cpp -o prog
```

This will take all your cpp files in the current directory and compile an executable named `prog`. (If you don't like `prog` you can choose any other name.)

You can automate the testing by entering test data into a file, say you called it `stdin.txt` (you can use any name you like):

```
0 1 2
0 -1 2
0 4 7
0 15 10
```

To send the input from `stdin.txt` to your program (instead of wasting time typing the test data again and again), do this:

```
./prog < stdin.txt
```

You can also send the output not to your terminal window but to a file like this:

```
./prog < stdin.txt > stdout.txt
```

You can now open `stdout.txt` to see the output.

Yet another level of software testing automation is to figure out the correct output and type the correct output to another file, say `correct.txt`. You can then get Fedora to test if `stdout.txt` is the same as `correct.txt` using this command:

```
diff correct.txt stdout.txt
```

If both files are the same, nothing is printed to your terminal window.

Hence your code–test cycle involves the following linux commands:

```
g++ *.cpp -o prog
./prog < stdin.txt > stdout.txt
diff correct.txt stdout.txt
```

(Of course you have to continually add test inputs to `stdin.txt` and the correct output to `correct.txt`.)

Operators

Operators are easy to understand. They are just functions except that you call them in a different way. Let me give you an example. First run the following example:

```
#include <iostream>


struct Blah
{
    int x;
    int y;
};


double funnyOperator(const Blah & i, const Blah & j)
{
    double d;
    d = (double)i.x / i.y + (double)j.x / j.y;
    return d;
}


int main()
{
    Blah b = {2, 3};
    Blah c = {5, 9};
    double d = funnyOperator(b, c);
    std::cout << d << std::endl;
    return 0;
}
```

(The actual body of the function is not important. Focus on how the function is called.) There are no surprises here. Now try this:

```
#include <iostream>


struct Blah
{
    int x;
    int y;
};


double operator+(const Blah & i, const Blah & j)
{
    double d;
    d = (double) i.x / i.y + (double) j.x / j.y;
    return d;
}


int main()
{
    Blah b = {2, 3};
    Blah c = {5, 9};
    double d = operator+(b, c);
    std::cout << d << std::endl;
    return 0;
}
```

The name of the function, `funnyOperator` is changed to `operator+`.

And finally run this program:

```
#include <iostream>


struct Blah
{
    int x;
    int y;
};


double operator+(const Blah & i, const Blah & j)
{
    double d;
    d = (double) i.x / i.y + (double) j.x / j.y;
    return d;
}
```

```
int main()
{
    Blah b = {2, 3};
    Blah c = {5, 9};
    double d = b + c;
    std::cout << d << std::endl;
    return 0;
}
```

As you can see, in the above program:

```
b + c
```

is really a call to the "function" `operator+`:

```
operator+(b, c)
```

except that the "function" `operator+`, in this context, is really called an **operator**. Once again, the following are actually the same:

```
b + c          operator+(b, c)
```

You can define all kinds of operators in C++, including

```
>>    <<    +    -    *    /
```

etc.

A simple `operator<<` is already defined for you which is why you can print a `Fraction` variable `r` with

```
std::cout << r;
```

Q1. `operator<<`

Modify `operator<<` to satisfy the following software requirements. I've deliberately not attempted to list the requirements in the most organized and efficient way so that you have to think through it yourself.

Negative signs are printed correctly, i.e., either not at all or once and never twice. For instance, if the `Fraction` variable `x` has `x.n = -2` and `x.d = -4`,

```
Fraction x = {-2, -4};
std::cout << x << std::endl;
```

will display

```
2/4
```

instead of

```
-2/-4
```

There should not be any negative sign for the printout of the denominator. For instance, if the `Fraction` variable `x` has `x.n = 5` and `x.d = -2`,

```
Fraction x = {5, -2};
std::cout << x << std::endl;
```

will display

```
-5/2
```

instead of

```
5/-2
```

If the denominator is 1 or -1, only an integer value is printed. For instance, if the `Fraction` variable `x` has `x.n = -3` and `x.d = -1`,

```
Fraction x = {-3, -1};
std::cout << x << std::endl;
```

will display

```
3
```

instead of

```
-3/-1
```

If the numerator is 0 (there is one exception: see below), then `0` is printed. In other words

```
Fraction x = {0, -5000};
std::cout << x << std::endl;
```

will display

```
0
```

If the denominator is 0, then `undefined` is printed. In other words

```
Fraction x = {123, 0};
std::cout << x << std::endl;
```

will display

```
undefined
```

Note that if the numerator and denominator are both 0, then `undefined` is printed.

Your `main()` should look like this:

```
#include <iostream>
#include "Fraction.h"


void test_print()
{
    int n = 0, d = 0;
    std::cin >> n >> d;
    Fraction f = {n, d};
    std::cout << f << std::endl;
}


int main()
{
    int option;
    std::cin >> option;

    switch (option)
    {
        case 1:
            test_print();
            break;
    }

    return 0;
}
```

Here are some test cases for you:

Test 1

```
1 1 3
1/3
```

Test 2

```
1 -1 3
-1/3
```

Test 3

```
1 1 -3
-1/3
```

Test 4

```
1 -12 -16
3/4
```

Test 5

```
1 0 3
0
```

Test 6

```
1 10 0
undefined
```

Add more test cases to your `stdin.txt` to go through all the above cases. I very strongly advise you to add as many test cases as you can and test your code as thoroughly as possible.

Q2. `operator+`

Define `operator+` so that when `operator+` is called with two `Fraction` variables, the return value is a `Fraction` value that models the addition of two fractions in the "real" world. Mathematically, this is how you add two fractions:

$$\frac{a}{b} + \frac{c}{d} = \frac{ad + bc}{bd}$$

Note that `operator+` accepts two `Fraction` values and returns a `Fraction` value. Therefore, you have to add this to the `Fraction` header file:

```
// Author:
// Date  :
// File  : Fraction.h

#ifndef FRACTION_H
#define FRACTION_H

struct Fraction
{
    int n; // numerator
    int d; // denominator
};

std::ostream & operator<<(std::ostream &, const Fraction &);
Fraction operator+(const Fraction &, const Fraction &);

#endif
```

And the following test code to your `main()`:

```
// Author:
// Date  :
// File  : test_fraction.cpp

#include <iostream>
#include "Fraction.h"


void test_print()
{
    int n = 0, d = 0;
    std::cin >> n >> d;
    Fraction f = {n, d};
    std::cout << f << std::endl;
}


void test_add()
{
    int n0, d0, n1, d1;
    std::cin >> n0 >> d0 >> n1 >> d1;
    Fraction r0 = {n0, d0};
    Fraction r1 = {n1, d1};
    std::cout << (r0 + r1) << std::endl;
}


int main()
{
    int option;
    std::cin >> option;

    switch (option)
    {
        case 1:
            test_print();
            break;
        case 2:
            test_add();
            break;
    }

    return 0;
}
```

Here are a couple of test cases for you:

Test 1

```
2 1 3 2 4
5/6
```

Test 2

```
2 5 7 -2 13
```

```
51/91
```

Add more test cases to your `stdin.txt`.

Hint: Spoilers ahead ... read only if you really need it ...

Mathematically this is how we add fractions:

$$\frac{a}{b} + \frac{c}{d} = \frac{ad + bc}{bd}$$

If `x` and `y` are the parameters in the body of `operator+`, then conceptually in terms of the members of `x` and `y`, we have:

$$\texttt{x + y} = \frac{\texttt{x.n}}{\texttt{x.d}} + \frac{\texttt{y.n}}{\texttt{y.d}} = \frac{\texttt{(x.n) * (y.d) + (x.d) * (y.n)}}{\texttt{(x.d) * (y.d)}}$$

The operator looks like this:

```
Fraction operator+(const Fraction & x, const Fraction & y)
{
    Fraction sum;
    return sum;
}
```

Of course you need to set the numerator and denominator of the value to return:

```
Fraction operator+(const Fraction & x, const Fraction & y)
{
    Fraction sum = {???, ???};
    return sum;
}
```

Q3. `operator-`

Once you've understood `operator+`, this part is easy.

Define `operator-` so that when `operator-` is called with two `Fraction` variables, the return value is a `Fraction` value that models the difference between the fractions in the "real" world. So what's the appropriate prototype?

Mathematically this is how you subtract two fraction numbers:

$$\frac{a}{b} - \frac{c}{d} = \frac{ad - bc}{bd}$$

You obviously need the following prototype:

```
Fraction operator-(const Fraction &, const Fraction &);
```

Add it to your `Fraction.h` and the following to your `main()`:

```
...
void test_subtract()
{
    int n0, d0, n1, d1;
    std::cin >> n0 >> d0 >> n1 >> d1;
    Fraction r0 = {n0, d0};
    Fraction r1 = {n1, d1};
    std::cout << (r0 - r1) << std::endl;
}


int main()
{
    int option;
    std::cin >> option;

    switch (option)
    {
        // other cases
        case 3:
            test_subtract();
            break;
    }

    return 0;
}
```

Here is a test case for you:

Test 1

```
3 2 -3 -2 5
-4/15
```

Add more test cases to your `stdin.txt`.

Q4. `operator*`

Define `operator*` so that when `operator*` is called with two `Fraction` variables, the return value is a `Fraction` value that models the product of the fractions in the "real" world. So what's the appropriate prototype?

Mathematically, this is how you multiply fractions:

$$\frac{a}{b} \cdot \frac{c}{d} = \frac{a \cdot c}{b \cdot d}$$

Don't forget to add the prototype to `Fraction.h` and the test code to your `main()`. This should be test option 4.

Test 1

```
4 2 3 11 17
22/51
```

Also, from now on, I won't provide you test cases. It's your job to add more test cases to your `stdin.txt` and test your code as thoroughly as possible.

Q5. `operator/`

Define `operator/` so that when `operator/` is called with two `Fraction` variables, the return value is a `Fraction` value that models the division of the first fraction by the second fraction in the "real" world. So what's the prototype of this operator?

Mathematically, this is how you divide fractions:

$$\frac{a}{b} \Big/ \frac{c}{d} = \frac{a \cdot d}{b \cdot c}$$

Don't forget to add the prototype to `Fraction.h` and the test code to your `main()`. This should be option 5.

Add more test cases to your `stdin.txt`.

Q6. `operator==`

Note that if `x` and `y` are `Fraction` values, then

```
x == y
```

is the same as

```
operator==(x, y)
```

The prototype of `operator==()` is

```
bool operator==(const Fraction &, const Fraction &);
```

Define `operator==` so that when `operator==` is called with two `Fraction` variables, the return value is true exactly when the `Fraction` values have the same mathematical value.

Mathematically, two fractions

$$\frac{a}{b} \quad \text{and} \quad \frac{c}{d}$$

are the same, i.e.,

$$\frac{a}{b} = \frac{c}{d}$$

if

$$ad = bc \tag{A}$$

Note that this is *not* the same as

$$a = c \quad \text{and} \quad b = d \tag{B}$$

which is *incorrect*. For instance, the fractions $\frac{1}{2}$ and $\frac{2}{4}$ are the same which is correctly detected by condition (A) but not (B).

You are to print out the boolean value resulting from the comparison between fractions.

Add the prototype to `Fraction.h` and the following test code to `main()`:

```
void test_eq()
{
    int n0, d0;
    int n1, d1;
    std::cin >> n0 >> d0 >> n1 >> d1;
    Fraction f = {n0, d0};
    Fraction g = {n1, d1};
    std::cout << (f == g) << std::endl;
    return;
}
```

This is test option 6. Here is a test case:

Test 1

```
6 1 3 2 6
1
```

Add more test cases to `stdin.txt`.

Q7. `operator!=`

Now add `operator!=` to your code. Note that if `x` and `y` are `Fraction` values, then

```
x != y
```

is the same as

```
operator!=(x, y)
```

What is the prototype of this operator?

It's very important to note the following: Instead of implementing `operator!=` from scratch, your `operator!=` *must* use `operator==`, i.e., `operator!=` only needs to call `operator==`.

Add appropriate code to your `Fraction.h` and `main()` as needed. This is test option 7.

Test 1

```
7 1 3 2 6
0
```

Add more test cases to your `stdin.txt`.

[Hint: `operator!=` is just the "opposite" of `operator==`.]

Q8. `reduce()` function

Note that all the above "functions" are operators. For this part, you need to write a function. This function reduces the `Fraction` to its lowest terms. Mathematically, the fraction

$$\frac{18}{60}$$

is not a reduced (or simplified) fraction. That's because 2 is a factor of 18 and 60:

$$\frac{18}{60} = \frac{2 \cdot 9}{2 \cdot 30}$$

If 2 is removed from the numerator and denominator, we get

$$\frac{18}{60} = \frac{2 \cdot 9}{2 \cdot 30} = \frac{9}{30}$$

This is still not reduced since 3 is a factor of 9 and 30. Removing 3 we get

$$\frac{18}{60} = \frac{2 \cdot 9}{2 \cdot 30} = \frac{9}{30} = \frac{3 \cdot 3}{3 \cdot 10} = \frac{3}{10}$$

This fraction, $\frac{3}{10}$ is now reduced: you can't find any integer factor of both 3 and 10 other than 1 and -1.

For this part, you need to implement:

```
void reduce(Fraction &);
```

that reduces the reference parameter so that the fraction is in its reduced form. For instance,

```
Fraction r = {18, 60};
reduce(r);
// at this point, r.n = 3 and r.d = 10
```

It's not enough to just remove a single or two common divisor: you must remove **all** common divisors which are greater than 1. There are a couple of other points:

- The `reduce()` function must also ensure that the denomiator of the parameter is positive:

```
Fraction r = {1, -2};
reduce(r);
// at this point, r.n = -1 and r.d = 2

Fraction s = {-1, -2};
```

```
reduce(s);
// at this point, s.n = 1 and s.d = 2
```

- If the numerator is 0 and the denominator is not 0, then the denominator is set to 1:

```
Fraction r = {0, -1500};
reduce(r);
// at this point, r.n = 0 and r.d = 1
```

- If the denominator is 0 (the case where the fraction is not defined), the numerator is set to 1:

```
Fraction r = {-42,  0};
reduce(r);
// at this point, r.n = 1 and r.d = 0
```

Add test code and cases to the different files as necessary. Here's the test code:

```
void test_reduce()
{
    int n, d;
    std::cin >> n >> d;
    Fraction f = {n, d};
    reduce(f);
    std::cout << f << std::endl;
    return;
}
```

This is test option 8.

Test 1

```
8 18 60
3/10
```

[Hint: If d divides b, then, For instance, 2 divides 18 and 60. Therefore,

$$\frac{a}{b} = \frac{a/d}{b/d}$$

For instance, 2 divides 18 and 60. Therefore,

$$\frac{18}{60} = \frac{18/2}{60/2} = \frac{18/2}{60/2} = \frac{9}{30}$$

Your code should find the greatest common divisors g of the numerator and the denominator of a fraction, and then divide the numerator and denominator of this fraction by g. For this assignment, you need not worry about efficiency.]

Initializer, Get and Set functions

An important principle in Software Engineering is "**information hiding**".

The concept of numerator and denominator is now contained in the concept of `Fraction`. As much as possible, we want to prevent users from directly accessing the `n` and `d` member variables. Why? If users of our library minimize their access to member variables, then we can change the way we design the concept. For instance, suppose that instead of using two integers as member variables, suppose I know that the numerators and denominators of all the fractions I care about are positive and less than 1000. In that case, instead of modeling the fraction $\frac{2}{3}$ with

```
n = 2, d = 3
```

I can model it with **ONE** single integer

```
nd = 0002003
```

i.e., I use the lower–order 3 digits of a single integer to model the denominator and the higher–order 3 digits to model the numerator. The highest–order digit models the sign. This saves me 50% of memory for every `Fraction` variable. In this case, my `Fraction struct` would look like

```
struct Fraction
{
    int nd;
};
```

and I have to change the body of the operators and functions accordingly.

If users do NOT touch member variables, their code need NOT change a single bit if I made the change to this new `struct`. This technique therefore allows software engineers to work independently without stepping on each other's toes. It's a technique for containing software change.

But for me to have this freedom, I do NOT want users of my `Fraction struct` to touch the member variables. Therefore, I must provide ways to read and write/modify the numerator and denominator of `Fraction` variables without accessing them directly. The following functions are therefore needed:

```
Fraction get_Fraction(int n, int d); // Used to create a Fraction
                                     // with the given n and d
int get_n(const Fraction & f);       // Return copy of the numerator
int get_d(const Fraction & f);       // Return copy of the denominator
void set_n(Fraction & r, int new_n); // Set r.n to new_n
void set_d(Fraction & r, int new_d); // Set r.d to new_d
void set(Fraction & r, int new_n, int new_d); // Sets both n and d of r
```

```
                                                // with the new values
```

(Why are some “pass by reference” while others “pass by constant reference”?)

To make these functions concrete to you, here are examples of how these functions can be used:

```
Fraction f = get_Fraction(7, 5); // f.n is 7 and f.d is 5

std::cout << get_n(f) << '\n'; // 7 is printed
std::cout << get_d(f) << '\n'; // 5 is printed

set_n(f, 100);                  // f.n becomes 100
std::cout << get_n(f) << '\n'; // 100 is printed
std::cout << get_d(f) << '\n'; // 5 is printed

set_d(f, -23);                  // f.d becomes -23
std::cout << get_n(f) << '\n'; // 100 is printed
std::cout << get_d(f) << '\n'; // -23 is printed

set(f, 22, 7);                  // f.n becomes 22 and f.d becomes 7
std::cout << get_n(f) << '\n'; // 22 is printed
std::cout << get_d(f) << '\n'; // 7 is printed
```

Once this is done, you should look at your test code and make sure it minimizes access to member variables. For instance, this line in `test_fraction.cpp`:

```
...
           Fraction r = {n, d};
...
```

can be changed to

```
...
           Fraction r = get_Fraction(n, d);
...
```

This will allow your `get_Fraction()` to set the numbers in whatever way you like, instead of exposing the fact that it’s a struct with two values.

By the way, functions for initializing, getting, and setting values in a `struct` variable are usually developed first.

Note that there is nothing to prevent the users from accessing the `n` and `d` member variables. You can tell them to use your initialier, getter, and setter functions, but they can still go behind your back. Later, when we talk about classes and objects, we'll see how to strictly enforce information hiding to protect the internal representation of objects.

Don't simplify the fraction for any of these functions. If you get a $\frac{2}{-0}$, store it as a `Fraction` with `n = 2` and `d = -0`.

Q9. `get_Fraction()` function

Implement `get_Fraction()` such that the following:

```
get_Fraction(n, d);
```

returns a `Fraction` value with numerator `n` and denominator `d`.

Add test code to your `main()` to take `n` and `d` and print the `Fraction` returned by `get_Fraction()`. This should be option 9.

Here are a few test cases:

Test 1

```
9 14 13
14/13
```

Test 2

```
9 12 -4
-3
```

Test 3

```
9 -2 -0
undefined
```

Add more test cases to your `stdin.txt`.

Q10. `get_n()` function

Implement `get_n()` such that the following:

```
get_n(f);
```

returns the numerator of the `Fraction` value `f`.

Add test code to your `main()` (This is option 10.)

Here are a few test cases:

Test 1

```
10 14 13
14
```

Test 2

```
10 11 12
11
```

Test 3

```
10 -2 -0
-2
```

Add more test cases to your `stdin.txt`.

Q11. `get_d()` function

Implement `get_d()` such that the following:

```
get_d(f);
```

returns the denominator of the `Fraction` value `f`.

Add test code to your `main()`. (This is option 11.)

Here are a few test cases:

Test 1

```
11 14 13
13
```

Test 2

```
11 11 12
12
```

Test 3

```
11 -2 -0
0
```

Add more test cases to your `stdin.txt`.

Q12. `set_n()` function

Implement `set_n()` such that the following:

```
set_n(f, n);
```

changes the numerator of the `Fraction` value `f` to `n`.

Add test code to your `main()`. If you think about it, you have to call `get_n()` after `set_n()` to check that it worked. So do that.

Here are a few test cases:

Test 1

```
12 17
17
```

(Create a fraction and set the numerator to 17.)

Test 2

```
12 1
1
```

Test 3

```
12 3
3
```

Add more test cases to your `stdin.txt`.

Q13. `set_d()` function

Implement `set_d()` such that the following:

```
set_d(f, d);
```

changes the denominator of the `Fraction` value `f` to `d`.

Add test code to your `main()`. This is test option 13.

Here is a test case:

Test 1

```
13 -18
-18
```

Add more test cases to your `stdin.txt`.

Q14. `set()` function

Implement `set()` such that the following:

```
set(f, n, d);
```

changes the numerator of the `Fraction` value `f` to `n` and the denominator of `f` to `d`.

Here's the test code to your `main()`:

```
void test_set()
{
    Fraction f;
    int n, d;
    std::cin >> n >> d;
    set(f, n, d);
    std::cout << f << std::endl;
}
```

(This is option 14.)

Here is a test case:

Test 1

```
14 17 -18
-17/18
```

Add more test cases to your `stdin.txt`.

Type conversion functions

Frequently types don't work alone. You usually want to convert data from one form to another. The most obvious features we want for our `Fraction` library is to work with integers and doubles.

Here are some of the ways we want our functions to work:

```
Fraction f = get_Fraction(1, 2);

double d = get_double(f);             // d has value 0.5
int i = get_int(get_Fraction(12, 5)); // i has value 2, essentially the
                                      // int value of 12.0/5
```

What are the appropriate prototypes for the `get_double()` and `get_int()` functions?

Q15. `get_int()` function

Implement `get_int()`.

Here's the test code to be added to your main C++ source file:

```
void test_get_int()
{
    int n, d;
    std::cin >> n >> d;
    std::cout << get_int(get_Fraction(n, d)) << std::endl;
    return;
}
```

This is test option 15.

Test 1

```
15 3 2
1
```

Q16. `get_double()` function

Implement `get_double()`. Calling `get_double` on `Fraction(1, 2)` should give you the `double` `0.5`.

Remember to test it in your `main()`. This is test option 16.

Test 1

```
16 3 2
1.5
```

Q17. `operator+=`

Write a function

```
Fraction & operator+=(Fraction &, const Fraction &);
```

so that you can execute the following code:

```
Fraction f = get_Fraction(1, 4);
Fraction g = get_Fraction(1, 2);

f += g; // increments f by g
std::cout << f << std::endl;

(f += g) += g; // increments f by g twice
std::cout << f << std::endl;
```

Here's the test code:

```
void test_addeq()
{
    int n0, d0;
    int n1, d1;
    std::cin >> n0 >> d0;
    std::cin >> n1 >> d1;
    Fraction f = get_Fraction(n0, d0);
    Fraction g = get_Fraction(n1, d1);
    f += g;
    std::cout << f << std::endl;
}
```

This is test option 17.

Test your code!

Q18. `operator-=`

Write a function

```
Fraction & operator-=(Fraction &, const Fraction &);
```

So that you can execute the following code:

```
Fraction f = get_Fraction(1, 4);
Fraction g = get_Fraction(1, 2);

f -= g; // same effect as f = f - g
std::cout << f << std::endl;

(f -= g) -= g; // same effect as f = f - g followed by f = f - g
std::cout << f << std::endl;
```

This is test option 18.

Test your code!

Test 1

```
18 2 -3 -2 5
-4/15
```

Q19. `operator*=`

Write a function

```
Fraction & operator*=(Fraction &, const Fraction &);
```

So that you can execute the following code:

```
Fraction f = get_Fraction(1, 4);
Fraction g = get_Fraction(1, 2);

f *= g; // same effect as f = f * g
std::cout << f << std::endl;

(f *= g) *= g; // same effect as f = f * g followed by f = f * g
std::cout << f << std::endl;
```

Test your code!

Test 1

```
19 2 3 11 17
22/51
```

Q20. `operator/=`

Write a function

```
Fraction & operator/=(Fraction &, const Fraction &);
```

So that you can execute the following code:

```
Fraction f = get_Fraction(1, 4);
Fraction g = get_Fraction(1, 2);

f /= g; // same effect as f = f / g;
std::cout << f << std::endl;

(f /= g) /= g; // same effect as f = f / g followed by f = f / g
std::cout << f << std::endl;
```

Here's the test code:

```
void test_diveq()
{
    int n0, d0;
    int n1, d1;
    std::cin >> n0 >> d0;
    std::cin >> n1 >> d1;
    Fraction f = get_Fraction(n0, d0);
    Fraction g = get_Fraction(n1, d1);
    f /= g;
    std::cout << f << std::endl;
}
```

This is test option 20.

Coda and miscellaneous notes

You're done with the assignment. Here are some comments. (You do not need to implement any of the features mentioned below.)

The above implies that you can, for instance, given a polynomial with fractional coefficients, you now find the exact fractional roots in a range and up to an accuracy of, say 1/100. For instance to find fractional roots in [0, 100] of

$$\frac{1}{3}x^3 - \frac{5}{3}x + \frac{7}{42} = 0$$

you do

```
Fraction start = get_Fraction(0, 1);
Fraction end = get_Fraction(100, 1);
Fraction step = get_Fraction(1, 100);

Fraction a = get_Fraction(1, 3);
Fraction b = get_Fraction(3, 3);
Fraction c = get_Fraction(7, 42);
Fraction zero = get_Fraction(0, 1);

for (Fraction x = start; x <= end; x += step)
{
    reduce(x);
    if (a * x * x * x - b * x + c == zero)
    {
        std::cout << x << std::endl;
    }
}
```

Note that there are several convenient features which are still missing in your `Fraction` library.

You do not have mixed mode operations. For instance you cannot perform addition of a fraction and an int:

```
Fraction f = get_Fraction(1, 3);
f = f + 2; // NOT POSSIBLE
```

Right now, we add two `Fraction` values. We have not taught our library how to add a `Fraction` with an `int` or an `int` with a `Fraction`. To do this, you need to implement the following:

```
Fraction operator+(const Fraction &, int);
```

This also appears in the above:

```
...
    if (a * x * x * x - b * x + c == zero)
...
```

You do not have `operator==` that compares a `Fraction` value with an int. If you do have

```
bool operator==(const Fraction &, int);
```

then the above boolean expression simplies to

```
...
    if (a * x * x * x - b * x + c == 0)
...
```

Etc.

The issue of converting a `double` to a `Fraction` also requires more thought. For instance, the `double` 0.333333333 is probably meant to be the fraction $\frac{1}{3}$. However the type conversion function from `double` to `Fraction` might be written so that the fraction is $\frac{333333333}{11000000000}$.