Displaying an interpolating line

Consider the following problem:

> Write a function that takes four arguments representing two points: (x_1, y_1) and (x_2, y_2).
> It will return nothing, but it will print the line that passes through these two points.
> You may assume that $x_1 \neq x_2$.

Sample required output is as follows, where the formatting of the double-precision floating-point numbers is as it is for `std::cout` printing each `double`:

```
y = 2.35432x + 4.54312
y = -53.2534x - 9.59341
y = 13.2349x
y = x + 3.14323
y = -x + 1.5
y = -8.18323
y = 0
```

You will note that terms with a zero coefficient are not printed unless the entire polynomial is the zero polynomial, in which case just `0` is printed. Also note that, for example, if the coefficient of the leading term is negative, the negative sign and the coefficient are juxtaposed, but if it is a subsequent term in the polynomial, the minus sign appears in place of the plus sign.

The function declaration will be:

```
void print_interpolating_line( double x1, double y1, double x2, double y2 );
```

The output will require us to find the coefficients a and b, and then print the equation.

We should break the problem into two parts:

1. One function will find the coefficients a and b.
2. Another function will print the linear equation.

Let us deal with the easier problem first.

Printing the line *ax* + *b*

If you were to start with

```
void print_linear_equation( double a, double b ) {
    std::cout << "y = " << a << "x + " << b << std::endl;
}
```

If you were to test the above sample expressions, however, you would get:

```
y = 2.35432x + 4.54312
y = -53.2534x + -9.59341
y = 13.2349x + 0
y = 1x + 3.14323
y = -1x + 1.5
y = 0x + -8.18323
y = 0x + 0
```

Clearly, something is wrong with such an easy implementation. Try writing such a function that correctly prints out such an equation, and then go to the next page to see our solution. Your function will require a number of conditions based on the values of a and b.

Hopefully you tried this yourself first. Here is a solution that covers all cases:

```
void print_linear_equation( double a, double b ) {
    if ( a == 0 ) {
        std::cout << "y = " << b;
    } else {
        if ( a == 1.0 ) {
            std::cout << "y = x ";
        } else if ( a == -1.0 ) {
            std::cout << "y = -x ";
        } else {
            std::cout << "y = " << a << "x ";
        }

        if ( b < 0 ) {
            std::cout << "- " << (-b);
        } else if ( b > 0 ) {
            std::cout << "+ " << b;
        }
    }

    std::cout << std::endl;
}
```

We should test this function with *a* being negative, negative one, zero, one and positive; and with *b* being negative, zero and positive; and that will cover all possibilities. The expected outputs are in the comments.

```
int main() {
    print_linear_equation(  2.05686,  2.68444 );  // y = 2.05686x + 2.68444
    print_linear_equation(  6.28584,  0.0     );  // y = 6.28584x
    print_linear_equation(  7.21418, -8.30745 );  // y = 7.21418x - 8.30745
    print_linear_equation(  1.0,      7.18926 );  // y = x + 7.18926
    print_linear_equation(  1.0,      0.0     );  // y = x
    print_linear_equation(  1.0,     -2.98002 );  // y = x - 2.98002
    print_linear_equation(  0.0,      7.18926 );  // y = 7.18926
    print_linear_equation(  0.0,      0.0     );  // y = 0
    print_linear_equation(  0.0,     -2.98002 );  // y = -2.98002
    print_linear_equation( -1.0,      7.18926 );  // y = -x + 7.18926
    print_linear_equation( -1.0,      0.0     );  // y = -x + 0
    print_linear_equation( -1.0,     -2.98002 );  // y = -x - 2.98002
    print_linear_equation( -6.29294,  7.05365 );  // y = -6.29294x + 7.05365
    print_linear_equation( -2.72300,  0.0     );  // y = -2.72300x
    print_linear_equation( -3.35002, -6.42892 );  // y = -3.35002x - 6.42892

    return 0;
}
```

Once this function is tested, the functionality for finding *a* and *b* can be implemented independently.

Finding the interpolating line

Given the two points (x_1, y_1) and (x_2, y_2), for the line passing through these points, it is necessary that

$$\begin{aligned} ax_1 + b &= y_1 \\ ax_2 + b &= y_2 \end{aligned}$$

Adding –1 times Equation 1 onto Equation 2, we get

$$\begin{aligned} ax_1 + b \quad &= y_1 \\ ax_2 - ax_1 &= y_2 - y_1 \end{aligned}$$

Solving this for a yields

$$a = \frac{y_2 - y_1}{x_2 - x_1}.$$

In C++, this is the statement

```
    a = (y2 - y1)/(x2 - x1);
```

Now, we could substitute this into Equation 1 to get

$$b = \frac{x_2 y_1 - x_1 y_2}{x_2 - x_1},$$

which would be the C++ statement

```
    b = (y2*x1 - y1*x2)/(x2 - x1);   // Statement 1
```

however, consider that once we have calculated a, we could just solve Equation 1 for b:

```
    b = y1 - a*x1;                   // Statement 2
```

In one way, Statement 1 is better than Statement 2, as it provides an independent calculation of b, so that if there was a mistake in calculating a, the formula for b will not also be incorrect; however, Statement 2 requires both fewer calculations and is much simpler: it's much less likely that you will make a mistake in writing down the second formula; indeed, if you did not already notice, Statement 1 is actually wrong.

Thus, using Statement 2, our function would be:

```
void print_interpolating_line( double x1, double y1, double x2, double y2 ) {
    assert( x1 != x2 );
    double a{ (y2 - y1)/(x2 – x1) };
    double b{ y1 - a*x1 };

    print_linear_equation( a, b );
}
```

Note that we included an assertion that the two *x*-values were different. We didn't need to include this, as the problem statement assured us that this was not necessary, but it doesn't hurt to check.

To test this, it's probably sufficient to test

```
print_interpolating_line( 1.0, 1.0, 2.0,  3.5 );  // a >  0, b < 0
print_interpolating_line( 1.0, 1.0, 2.0,  2.0 );  // a =  1, b = 0
print_interpolating_line( 1.0, 1.0, 2.0,  1.0 );  // a =  0, b > 0
print_interpolating_line( 1.0, 1.0, 2.0,  0.0 );  // a = -1, b > 0
print_interpolating_line( 1.0, 1.0, 2.0, -0.5 );  // a <  0, b > 0

print_interpolating_line( 1.0, -1.0, 2.0, -3.0 );  // a <  0, b > 0
print_interpolating_line( 1.0, -1.0, 2.0, -2.0 );  // a = -1, b = 0
print_interpolating_line( 1.0, -1.0, 2.0, -1.0 );  // a =  0, b < 0
print_interpolating_line( 1.0, -1.0, 2.0,  0.0 );  // a =  1, b < 0
print_interpolating_line( 1.0, -1.0, 2.0,  0.5 );  // a >  0, b < 0

print_interpolating_line( 1.0,  0.0, 2.0,  0.0 );  // a = 0, b = 0
```

Why break the problem into two?

You may be wondering why we are breaking what seems to be a straight-forward problem into two sub-problems: finding *a* and *b*, and then printing the linear equation. First and foremost is testing. Look at the tests that were required to try to test all possible outputs for the `print_linear_equation(…)` function. If you had to come up with fifteen times four or 60 different *x* and *y* values so these fifteen sets produced all of these conditions (positive, one, zero, negative one and negative slope; and positive, zero and negative *y*-intercept), this would likely be frustrating and you would be as likely to make a mistake in authoring the tests as you would have been authoring the solution itself. If you look at the tests for solving the system of linear equations, you will see that they focus not on all the variations of possible outputs, but rather they focus on finding different interpolating lines.

The second benefit is that if you combined everything into one function, you may have bugs in one part of the code that may appear to be the result of a bug somewhere else.

```
void print_interpolating_line( double x1, double y1, double x2, double y2 ) {
    double a{ (y2 - y1)/(x2 - x2) };
    double b{ y1 - a*x1 };

    if ( a = 0 ) {
        std::cout << "y = " << b << std::endl;
    } else {
        if ( a == 1.0 ) {
            std::cout << "y = x ";
        } else if ( a == -1.0 ) {
            std::cout << "y = -x ";
        } else {
            std::cout << "y = " << a << "x ";
        }

        if ( b < 0 ) {
            std::cout << "- " << (-b) << std::endl;
        } else {
            std::cout << "+ " << b << std::endl;
        }
    }
}
```

This implementation, however, has a bug in it. If you call this function:

```
print_interpolating_line( 0.1, 0.7, 1.6, 1.3 );
```

the output is

```
y = 0x + 1.65
```

This is clearly incorrect. Where is the error? You may think the error is in the calculation of *a* and *b*, but that is not actually the case.

By breaking the problem into two sub-problems, by thoroughly testing the second before even attempting the first, you are guaranteed that if there is an error in the subsequent implementation of finding the coefficients of the interpolating polynomial, then that is exactly where the error is. The error above, in case you did not find it, is not in the calculation of the coefficients of a and b, but rather in the printing of the equation. What is the first condition? What should it be?

Online

Once you have tried this yourself, you can look at the complete solution at `repl.it` with the URL

https://repl.it/@dwharder/Displaying-an-interpolating-line.