Printing a quadratic polynomial

Consider the following problem:

> Write a function that takes three arguments, the coefficients of $ax^2 + bx + c$.
> It will return nothing, but it will print the quadratic equation to the screen in the form of $(x - r_1)(x - r_2)$, $(x - r)^2$, $x - r$ and c if there are two different (and possibly complex) roots, a double root, one root or a constant polynomial, respectively.

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 = -(x - 2.35432)(x + 4.54312)
y = 3.25(x - 53.2534)(x - 9.59341)
y = -0.5(x - 13.2349)
y = (x - 13.2349)^2
y = -3.5x^2
y = (x + 3.14323 - 4.32531j)(x + 3.14323 + 4.32531j)
y = 6(x + 1 - 2j)(x + 1 + 2j)
y = -(x + 1 - 2j)(x + 1 + 2j)
y = 7.5(x + j)(x - j)
y = -8.18323
y = -(x - 3)
y = 5.23(x + 1)
y = x - 5.323
y = 0
```

It does not matter which root is printed first.

You will note that linear polynomials (when $a = 0$ but b is not zero) are printed without parentheses, and that constant polynomials (the special case when $a = 0$ and $b = 0$) simply print the constant coefficient c.

The function declaration will be:

```
void print_quadratic( double a, double b, double c );
```

The output will require us to find the roots (if any) and print them appropriately.

We should break the problem into two parts:

1. One function will find the roots (if any).
2. Another function will print $x - r$ for a given root r.

Let us deal with the easier problem first.

Printing the *x – r*

If you were to start with

```
void print_root( double r ) {
    std::cout << "x - " << r;
}
```

you would immediately note that this will not print a complex root.

```
void print_root( double re, double im ) {
    std::cout << "x - " << re << " – " << im << "j";
}
```

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

```
(x - 2.35432 - 0j)(x - -4.54312 - 0j)
(x - 53.2534 - 0j)(x - 9.59341 - 0j)
(x - 13.2349 - 0j)
(x - -3.14323 - 4.32531j)(x - -3.14323 - -4.32531j)
(x – 0 - 1j)(x – 0 - -1j)
```

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

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

```
void print_root( double re, double im ) {
    std::cout << "x";

    if ( re > 0.0 ) {
        std::cout << " - " <<  re;
    } else if ( re < 0.0 ) {
        std::cout << " + " << -re;
    }

    // Do not print the imaginary component if it is 1 or -1
    if ( im == 1.0 ) {
        std::cout << " - j";
    } else if ( im > 0.0 ) {
        std::cout << " - " <<  im << "j";
    } else if ( im == -1.0 ) {
        std::cout << " + j";
    } else if ( im < 0.0 ) {
        std::cout << " + " << -im << "j";
    }
}
```

Now, some of you who are more aware of the implementation of strings and characters may wonder why do we not use, for example,

```
        std::cout << " + " << -im << 'j';
```

You could do this, but it's a question of purpose: a character is a character in a string, while a string is generally thought of to be something to be printed (that is, containing text). Both work, but now any character is always used for the manipulation of strings, and strings are what are printed to the screen.

Now, looking at what is required, one may ask if this function should also print the parentheses. If $a = 0$ but b is non-zero, we do not require parentheses in any case, and if the root is zero, we do not require parentheses. The second we may deduce in this function (by checking if both parameters are zero), but the first requires information from the individual calling the function. Thus, we require another parameter, and we only print the parentheses if we are asked to and if either the real or imaginary components are non-zero:

```
void print_root( double re, double im, bool print_parentheses ) {
    if ( print_parentheses && ((re !== 0.0) || (im != 0.0)) ) {
        std::cout << "(";
    }

    std::cout << "x";

    if ( re > 0.0 ) {
        std::cout << " - " <<  re;
    } else if ( re < 0.0 ) {
        std::cout << " + " << -re;
    }

    // Do not print the imaginary component if it is 1 or -1
    if ( im == 1.0 ) {
        std::cout << " - j";
    } else if ( im > 0.0 ) {
        std::cout << " - " <<  im << "j";
    } else if ( im == -1.0 ) {
        std::cout << " + j";
    } else if ( im < 0.0 ) {
        std::cout << " + " << -im << "j";
    }

    if ( print_parentheses && ((re != 0.0) || (im != 0.0)) ) {
        std::cout << ")";
    }
}
```

We should test this function with *re* being negative, zero and positive; with *im* being negative, negative one, zero, one and positive; and with the requirement to print parentheses being both true and false; and that will cover all possibilities. The expected outputs are in the comments.

```
int main() {
    print_root(  2.05686,  2.68444, true );
    print_root(  2.05686,  1.0,     true );
    print_root(  2.05686,  0.0,     true );
    print_root(  2.05686, -1.0,     true );
    print_root(  2.05686, -4.50233, true );

    print_root(  0.0,      2.68444, true );
    print_root(  0.0,      1.0,     true );
    print_root(  0.0,      0.0,     true );
    print_root(  0.0,     -1.0,     true );
    print_root(  0.0,     -4.50233, true );

    print_root( -2.05686,  2.68444, true );
    print_root( -2.05686,  1.0,     true );
    print_root( -2.05686,  0.0,     true );
    print_root( -2.05686, -1.0,     true );
    print_root( -2.05686, -4.50233, true );

    print_root(  2.05686,  2.68444, false );
    print_root(  2.05686,  1.0,     false );
    print_root(  2.05686,  0.0,     false );
    print_root(  2.05686, -1.0,     false );
    print_root(  2.05686, -4.50233, false );

    print_root(  0.0,      2.68444, false );
    print_root(  0.0,      1.0,     false );
    print_root(  0.0,      0.0,     false );
    print_root(  0.0,     -1.0,     false );
    print_root(  0.0,     -4.50233, false );

    print_root( -2.05686,  2.68444, false );
    print_root( -2.05686,  1.0,     false );
    print_root( -2.05686,  0.0,     false );
    print_root( -2.05686, -1.0,     false );
    print_root( -2.05686, -4.50233, false );

    return 0;
}
```

Incidentally, you could consider that this may be used by mathematicians, who may want to use *i* instead of *j*. This would require one more parameter, but that is not in the requirements of this problem.

Once this function is tested, the functionality for finding the roots can be implemented independently.

Finding the roots

Given the polynomial $ax^2 + bx + c$, there may be two roots (possibly double), one root, or it is a constant polynomial.

First, if a = b = 0, then we need only print c. Thus, our output will be

```
std::cout << c;
```

Second, if $a = 0$ but b is not, then the root is found by solving $bx + c = 0$, or $-\frac{c}{b}$.

In this case, we will generate the appropriate output by calling

```
print_root( -c/b, 0.0, b != 1 ); // zero imaginary part,
                                 // don't print () if b == 1
```

If a is non-zero, the roots are found from the quadratic equation:

$$\frac{-b \pm \sqrt{b^2 - 4ac}}{2a}.$$

Working with this is actually sub-optimal, so instead, we will rewrite this as

$$-\frac{b}{2a} \pm \frac{\sqrt{b^2 - 4ac}}{2a}.$$

Now, everything depends on the discriminant $b^2 - 4ac$. Let assign this to d, so $d \stackrel{\text{def}}{=} b^2 - 4ac$; that is, d is being defined as $b^2 - 4ac$, or d is assigned the value of $b^2 - 4ac$. In C++, this would be the statement

```
double disc{ b*b - 4.0*a*c }; // We will give the local variable a clearer name
```

Next, if $d > 0$, there are two real roots, if $d = 0$, there is a double root, and if $d < 0$, there are two complex roots. Thus, in these three cases, we should be printing to the screen:

$$a\left(x - \left(-\frac{b}{2a}\right) - \frac{\sqrt{b^2 - 4ac}}{2a}\right)\left(x - \left(-\frac{b}{2a}\right) + \frac{\sqrt{b^2 - 4ac}}{2a}\right),$$

$$a\left(x - \left(-\frac{b}{2a}\right)\right)^2, \text{ and}$$

$$a\left(x - \left(-\frac{b}{2a}\right) - \frac{\sqrt{4ac - b^2}}{2a}\, j\right)\left(x - \left(-\frac{b}{2a}\right) + \frac{\sqrt{4ac - b^2}}{2a}\, j\right),$$

respectively.

Fortunately, if we assign d as indicated before and define $r \stackrel{\text{def}}{=} -\frac{b}{2a}$, these simplify to

$$\left(x - r - \frac{\sqrt{d}}{2a}\right)\left(x - r + \frac{\sqrt{d}}{2a}\right),$$

$$(x - r)^2, \text{ and}$$

$$\left(x - r - \frac{\sqrt{-d}}{2a}\,j\right)\left(x - r + \frac{\sqrt{-d}}{2a}\,j\right),$$

respectively.

Thus, our function would be

```
void print_quadratic( double a, double b, double c ) {
    if ( (a == 0.0) && (b == 0.0) ) {
        std::cout << c << std::endl;
    } else if ( a == 0.0 ) {
        if ( b == -1.0 ) {
            std::cout << "-";
        } else if ( b != 1.0 ) {
            std::cout << b;
        }

        // If b != 1, we must print brackets:  b(x + c/b)
        //            otherwise, we print      x + c/b
        print_root( -c/b, 0.0, b != 1.0 );
        std::cout << std::endl;             // We must still print a new line
    } else {
        assert( a != 0.0 );

        if ( a == -1.0 ) {
            std::cout << "-";
        } else if ( a != 1.0 ) {
            std::cout << a;
        }

        double nb_2a{ -b/2.0/a };
        double disc{ b*b - 4.0*a*c };

        if ( disc > 0.0 ) {
            // Both roots are real and different
            disc = std::sqrt( disc )/2.0/a;

            print( nb_2a + disc, 0.0, true );
            print( nb_2a - disc, 0.0, true );
            std::cout << std::endl;
        } else if ( disc == 0.0 ) {
            // There is a double root at -b/2a
            print( nb_2a, 0.0, true ); // Print parentheses unless nb_2a == 0
            std::cout << "^2" << std::endl;
        } else {
            assert( disc < 0.0 );
            // There are two complex conjugate roots
            disc = std::sqrt( -disc )/2.0/a;

            print( nb_2a,  disc, true );
            print( nb_2a, -disc, true );

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

Here are some tests:

```
int main() {
    print_quadratic( 1.0, 0.0,  1.0 );
    print_quadratic( 2.0, 0.0,  5.0 );
    print_quadratic( 1.0, 0.0,  0.0 );
    print_quadratic( 1.0, 0.0, -1.0 );
    print_quadratic( 1.0, 1.0, -1.0 );
    print_quadratic( 1.0, 2.0, 1.0 );
    print_quadratic( 1.0, -2.0, 1.0 );
    print_quadratic( 1.0, -2.0, 5.0 );
    print_quadratic( 0.0, -2.0, 5.0 );
    print_quadratic( 0.0, -1.0, 5.0 );
    print_quadratic( 0.0,  0.0, 5.0 );
    print_quadratic( 0.0,  1.0, 5.0 );

    return 0;
}
```

Why break the problem into two?

One of the greatest benefits of breaking this problem into two parts is that we only had to deal with all of the special cases once. Imagine if in all the above cases were incorporated into one large program: it would be hundreds of lines of code. 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_quadratic( double a, double b, double c ) {
    if ( (a == 0.0) && (b == 0.0) ) {
        std::cout << c << std::endl;
    } else if ( a == 0.0 ) {
        // Code to print b(x - (-b/2a))
    } else {
        assert( a > 0.0 );

        double nb_2a{ -b/2.0/a };
        double disc{ b*b - 4.0*a*c };

        if ( disc > 0.0 ) {
            // Both roots are real and different
            disc = std::sqrt( disc )/2.0/a;
            // Code to print a(x - (-b/2a) - disc)(x - (-b/2a) + disc)
        } else if ( disc == 0.0 ) {
            // There is a double root at -b/2a
            // Code to print a(x - (-b/2a))^2 or x^2
        } else {
            // There are two complex conjugate roots
            disc = std::sqrt( -disc )/2.0/a;
            // Code to print a(x - (-b/2a) - discj)(x - (-b/2a) + discj)
        }
    }
}
```

You would have multiple places were you could make exactly the same mistake, and if you made that mistake once, you'd probably make that same mistake everywhere. Then, even worse, if you found the error for one case, you'd have to remember that you made a similar mistake elsewhere; however, the worst of all situations is that someone else may be required to fix one bug, and that person could not be expected to realize you made the exact same mistake elsewhere. Thus, by authoring a separate function, if you did make one mistake, once you found that one mistake in one place, you'd fix it everywhere else, too.

Or three?

When writing out the multiplier for both linear and quadratic polynomials, we have the following two sippets of code:

```
// Printing out a solution to the linear equation bx + c
if ( b == -1.0 ) {
    std::cout << "-";
} else if ( b != 1.0 ) {
    std::cout << b;
}

// Printing out a solution to the quadratic equation ax^2 + bx + c
if ( a == -1.0 ) {
    std::cout << "-";
} else if ( a != 1.0 ) {
    std::cout << a;
}
```

You will note the code is the same. Suppose you made a change to one piece of code: logic suggests that whatever change you made, it should also apply in the other location; however, this would require incredible detail in comments, something no programmer would ever do.

Instead, as soon as you see yourself writing similar code again, consider changing it to a function:

```
void print_leading_multiplier( double a ) {
    if ( a == -1.0 ) {
        std::cout << "-";
    } else if ( a != 1.0 ) {
        std::cout << a;
    }
}
```

Now, both bodies of the conditional statement would call this one function with the appropriate argument:

```
} else if ( a == 0.0 ) {
    print_leading_multiplier( b );

    // If b != 1, we must print brackets:  b(x + c/b)
    //            otherwise, we print      x + c/b
    print_root( -c/b, 0.0, b != 1.0 );
    std::cout << std::endl;             // We must still print a new line
} else {
    assert( a != 0.0 );

    print_leading_multiplier( a );

    double nb_2a{ -b/2.0/a };
    double disc{ b*b - 4.0*a*c };
```

As a bonus, the name of the function is much clearer than the code it replaces.

Online

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

https://repl.it/@dwharder/Printing-a-quadratic-polynomial.