CISC 1600—Computer Science I
Spring, 2015

Programming Project #2: Solving Quadratic Equations
Date Due: Monday 2 March 2015

Write a program that prompts the user to enter the coefficients of a quadratic polynomial $ax^2 + bx + c$. It should then solve the polynomial equation

$$ax^2 + bx + c = 0.$$

You should use the quadratic formula to solve this equation; if you don't remember it, you'll find it later on in this handout.

In what most people consider to be the normal case, there are two real roots, as in the case $x^2-3x+2 = 0$. However, there is the possibility of

- only one real root,[1] as in the case $x^2 - 2x + 1 = 0$, and
- no real roots, as with $x^2 + 1 = 0$.

Since you can't control the input your users are giving you, think about what to do if $a = 0$. (Note that the equation is now a linear equation, provided that $b \neq 0$.)

A few considerations:

1. You are to do this using only the techniques found in Chapters 1 through 4 of the text. In other words, you don't need to read ahead to figure out how to do this.

2. Of course, you'll need to compute the square root of the discriminant. The standard library has a function `sqrt()` that will be pretty helpful here. Its prototype (which you get for free) is

   ```
   double sqrt(double x);
   ```

   The return value of `sqrt(blah)` is (not surprisingly) the square root of `blah`, assuing that `blah` is non-negative.[2]

3. I expect you to work on this in lab on 6 February and 13 February. I wouldn't surprised if you completed the assignment in these lab periods. However, for this to work, you *must* design and code your solution before you set foot into the lab. I will not let you approach a terminal until you show me that you have written out the program in advance.

4. I have made an executable version of this program for you to try out. The executable is available as the file

   ```
   ~agw/class/cs1/share/proj2/proj2-agw
   ```

[1] More precisely, it's a repeated real root of multiplicity two.

[2] More honestly, the `sqrt()` function doesn't really return the exact square root of its argument, but an approximation to same. But it's a *very* good approximation.

on the Departmental Linux machines. You should copy it to your working directory, so you can use it for reference's sake. Assuming that you've logged onto one of our machines (i.e., on `erdos` or one of the lab machines), `cd` yourself into your working directory and execute the command

```
cp ~agw/class/cs1/share/proj2/proj2-agw .
```

The dot is part of the command; it means "current working directory". Once you've copied it over, *please* try it out, so you can see what I consider to be good input/output behavior, not to mention how I handle various strange data values (as mentioned above).

5. As you know by now, it's a good idea to let a function carry out a well-encapsulated subtask. When done carefully, this can shorten the `main()` function; my `main()` is only 21 lines long. I used two subsidiary functions:

   - The function having prototype

     ```
     void solve_linear(double b, double c);
     ```

     solves a linear equation $bx + c = 0$, distinguishing between the cases $b \neq 0$ and $b = 0$. My version was 14 lines long.
   - The function having prototype

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

     solves a genuine quadratic equation $ax^2 + bx + c = 0$, "genuine" meaning that $a \neq 0$. Mine was 21 lines long.

   The main reason for using functions here (other than simply getting practice) is that each function (including the `main()` function) is small enough to understand easily; in particular, each function takes up less than a screenful of space.

   I have made a "stub version" of `proj2.cc`, appropriately named `proj2-stub.cc`, which I have put into the `proj2` "share directory" mentioned above. This file

   - takes care of the overall logic flow,
   - declares the two functions mentioned above, and
   - gives the "stub" implementations of these functions.

   I highly recommend that you copy same to your working directory, renaming it as `proj2.cc`. For fun, you should compile this file "as is," and see if you understand why it works the way it does, in its limited way. If you now fill in the blanks (which means implementing the input section and completing the implementation of the two functions mentioned above), you'll now have a working program.

   Big hint: Do one thing at a time. First, take care of the input phase (this should be pretty easy). Run the program, to make sure that this piece works right. Now, implement one of the two functions, after which you should run the program and test that the program works right so far. Finally, implement the other function, and do yet more testing.

6. You should do this project, in a subdirectory `proj2` of your `~/private/cs1` directory. See the Project 1 handout for further discussion, and adjust accordingly.

7. Here is a good collection of sample data to use.

a	b	c	Equation	Explanation
1	-3	2	$x^2 - 3x + 2 = 0$	Two real roots ($x = 1$ and $x = 2$)
2	-6	4	$2x^2 - 6x + 4 = 0$	Two real roots ($x = 1$ and $x = 2$)
1	-2	1	$x^2 - 2x + 1 = 0$	One double root ($x = 1$)
1	-2	4	$x^2 - 2x + 4 = 0$	No real roots
0	2	4	$2x + 4 = 0$	Linear equation, root $x = -2$
0	0	0	$0 = 0$	An identity
0	0	1	$1 = 0$	A contradiction

8. Use the `photo` program so you can have something to turn in. *Don't do this until the program is working!* The commands you should issue are as follows:

```
photo
cat proj2.cc
g++ -o proj2 proj2.cc
proj2
proj2
proj2
proj2
proj2
proj2
proj2
exit
```

   You'll be running `proj2` once for each test dataset. Once again, see the Project 1 handout if you're still unclear about `photo`.

Good luck!

For 10 points extra credit:

The classical quadratic formula is fine if one is doing exact arithmetic of real numbers. For our discussion, we'll think of a real number as being a decimal number an infinite number of places beyond the decimal point. Computers cannot represent such numbers with exact precision; they can only *approximate* them. The common standard is to approximate real numbers by *floating-point numbers*. For the purpose of our discussion, we can think of a floating-point number as being a number written in scientific notation (such as $0.6626068 \times 10^{-34}$ or 0.6022142×10^{24}, with a fixed number of digits after the decimal point (in these examples, six places) and a limited range for the exponent (say, from -44 to 38). You'll also notice that there's a zero before the decimal point and that the first place after the decimal point is nonzero; such floating-point numbers are said to be *normalized*. The common standard is known as IEEE 754, which specifies both single-precision and double-precision floating-point arithmetic (which might be implemented via `float` and `double`, respectively, by a C++ compiler).

What this all means is that floating-point arithmetic is inaccurate, compared to real arithmetic. For example, whereas the commutative laws (for addition and multiplication) hold in floating point, the associative law does not hold; neither does the distributive law. The inaccuracy is generally negligible, but it occasionally can cause bad things to happen.

One such example is *catastrophic cancellation*, which occurs when subtracting nearly equal quantities. Suppose we are using floating-point arithmetic having ten decimal digits' worth of accuracy. Let's take an example, taken from

`http://www.zipcon.net/ swhite/docs/math/loss_of_significance.html`

Let $x = 0.12345678912345678 90 - 0.1234567890$, and suppose that we approximate x on a machine using floating-point arithmetic with 10-digit accuracy. Our best approximation would be

$$\bar{x} = 0.1234567891 - 0.1234567890 = 0.0000000001 = 0.100000000 \times 10^{-9}.$$

But the real answer is 0.000000001234567890, which can be represented in floating point as $0.1234567890 \times 10^{-9}$. So we only have one digit of accuracy in our answer, even though the answer can be represented in floating point with ten digits' worth of accuracy. Putting this another way, the *relative error* of our approximation is

$$\left| \frac{x - \bar{x}}{x} \right| \doteq 0.18999,$$

which is about 19%.

Now how does all this apply to our quadratic equation?

```
sobolev@dsm:proj2$ proj2
Enter the coefficients of a quadratic polynomial a*x**2 + b*x +c:
  a? 1
  b? -20000
  c? 1.5e-9
Trying to solve the quadratic equation 1*x*x + -20000*x + 1.5e-09 == 0
Two roots, x = 20000 and x = 0
```

Let's double-check this answer, using Mathematica:

```
sobolev@dsm:proj2$ math
Mathematica 9.0 for Linux x86 (64-bit)
Copyright 1988-2013 Wolfram Research, Inc.

In[1]:=  x^2 - 20000 x + 1.5*10^-9

                -9              2
Out[1]= 1.5 10   - 20000 x + x

In[2]:= Solve[%==0, x]

                       -14
Out[2]= {{x -> 7.5 10   }, {x -> 20000.}}

In[3]:= Quit
```

So one computed solution ($x = 20000$) is correct, whereas the relative error in the other computed solution ($x = 0$) has a relative error of 100%!!

The problem here is that the values of a, b, and c were carefully chosen so that although $b^2 \neq b^2 - 4ac$ in real arithmetic, the floating-point values of b^2 and $b^2 - 4ac$ are the same (you should check this yourself). Hence the discriminant of this polynomial computes out as being b^2, and so

$$x_+ = \frac{-b + \sqrt{b^2 - 4ac}}{2a} \doteq \frac{-b + \sqrt{b^2}}{2a} = \frac{0}{2 \cdot 1} = 0$$

in floating point.

What to do? Note that in this case, the other root

$$x_- = \frac{-b - \sqrt{b^2 - 4ac}}{2a}$$

does *not* cause a cancellation error (since both $-b$ and $-\sqrt{b^2 - 4ac}$ are negative). We can calculate the other root by using the fact that the product of the roots *must* equal c/a, which follows from some simple algebra.[3] Thus we can compute

$$x_+ = \frac{c}{ax_-}$$

Of course, if $b < 0$, things will work in the opposite direction.

So (finally!) for ten points' worth of extra credit, implement this stable version of `solve_quadratic()`. Sample runs of the program look like the following:

```
sobolev@dsm:proj2$ proj2-stable
Enter the coefficients of a quadratic polynomial a*x**2 + b*x +c:
  a? 1
  b? -20000
  c? 1.5e-9
Trying to solve the quadratic equation 1*x*x + -20000*x + 1.5e-09 == 0
Using classical formula: Two roots, x = 20000 and x = 0
Using stable formula: Two roots, x = 20000 and x = 7.5e-14
sobolev@dsm:proj2$ proj2-stable
Enter the coefficients of a quadratic polynomial a*x**2 + b*x +c:
  a? 1
  b? 20000
  c? 1.5e-9
Trying to solve the quadratic equation 1*x*x + 20000*x + 1.5e-09 == 0
Using classical formula: Two roots, x = 0 and x = -20000
Using stable formula: Two roots, x = -7.5e-14 and x = -20000
```

For your test data, run your program on these two additional data sets.

Remark: I have made an executable that does both the classical and stable solution of the quadratic equation; its name is `proj2-agw-stable`, and it may be found in the class share directory. You would be well advised to copy same into your working directory if you're shooting for the extra credit.

[3] Indeed, since the two roots are x_+ and x_- (these are numbers, and not variables!), we have

$$\begin{aligned} ax^2 + bx + c = a(x - x_+)(x - x_-) &= a\big(x^2 - (x_+ + x_-)x + (x_+x_-)\big) \\ &= ax^2 - a(x_+ + x_-)x + (ax_+x_-). \end{aligned}$$

Now compare the constant terms to see that $c = ax_+x_-$, and so $x_+x_- = c/a$.