Assignment C Lab Book

# Week 3 – Lab C

Oct 2022

## Q1. Debugging

Suggest a solution to make the program execute correctly.

Solution:

while (std::cin >> value) {

switch (value) {

case 1: equals1++;

break;

case 2: equals2++;

break;

case 3: equals3++;

break;

default:;

}

}

std::cin.clear();

Sample Output:

A computer screen capture

Description automatically generated with medium confidence

### Test data:

1,2,3

### **Reflection:**

### Without the break the statement the code was compiled in fell swoop this produced wrong outputs, however the break statements instructed the compiler to pause at every Switch command before proceeding to the next switch command, with these commands the code generated the accurate output as required.

### Metadata:

N/A

### Further information:

N/A

## Q2. Bitwise

## Write a program to read four separate 32-bit integers (red, green, blue, and alpha) and encode them into a single 32-bit value. Output this 32-bit value. Verify that the results are correct by taking the 32-bit value and extracting and outputting the separate integers.

Solution (a): Encoding 4 separate 32-bit integers into a single 32-bit value.

int main(int, char\*\*) {

unsigned char r = 18;

unsigned char g = 12;

unsigned char b = 15;

unsigned char a = 19;

unsigned int newColour = (((unsigned int)r) << 24) + (((unsigned int)g) << 16) + (((unsigned int)b) << 8) + ((unsigned int)a);

cout << "new Colour: 0x" << hex << newColour << endl;

Sample Output:

A screenshot of a computer

Description automatically generated

Solution (b): Converting a single 32-bit value into 4 separate 32-bit integers.

unsigned int colour = 0x120c0f13;

unsigned char r = (unsigned char)((colour >> 24) & 0xff);

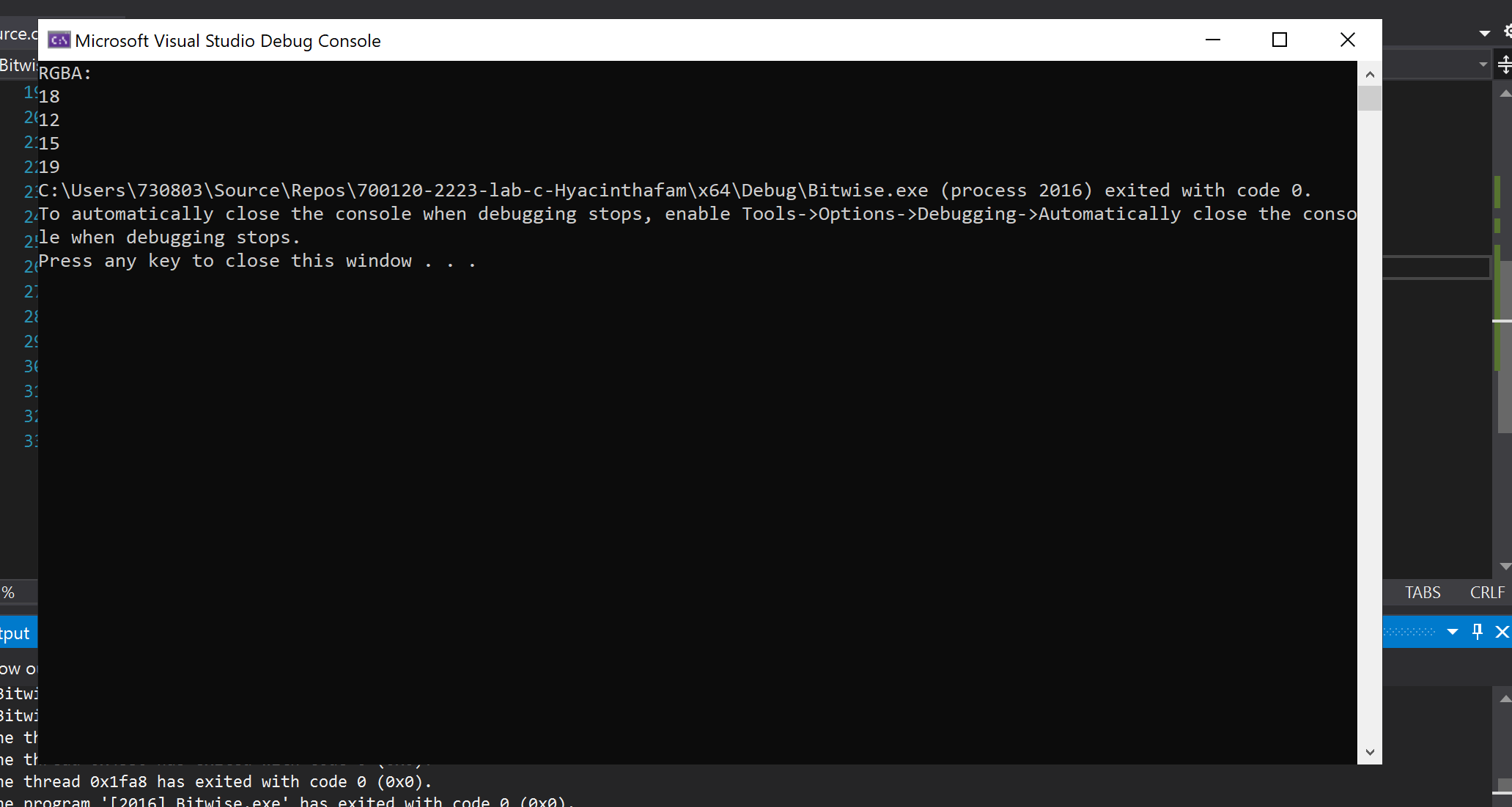
unsigned char g = (unsigned char)((colour >> 16) & 0xff);

unsigned char b = (unsigned char)((colour >> 8) & 0xff);

unsigned char a = (unsigned char)(colour & 0xff);

cout << "RGBA:" << endl << int(r) << endl << int(g) << endl << int(b) << endl << int(a); ;

Sample Output



Test Data:

N/A

Reflections:

This was fun to do, converting between single integer value and the exact 4 separate 32bit values was satisfying.

## Q3. Parsing

## Select and build the Parser project. This application scans through a C++ file looking for a particular variable. When and if it finds the variable it returns the word and line position of that variable. Comments are ignored. Familiarize yourself with this program and then modify the program to make the loop structures more efficient and easier to maintain.

## Q4. Quadratic

Select and build the Quadratic project.

Modify the quadratic program to:

1. Handle equal or imaginary roots

**Solution Equal Roots and Imaginary Roots:**

int main(int argc, char\*\* argv) {

std::cout << "Enter the coefficients for a quadratic equation" << std::endl;

double a, b, c, operand,imaginary\_root;

std::cin >> a >> b >> c;

if (a == 0) {

std::cout << "a cannot be zero" << std::endl;

}

else {

operand = b \* b - 4.0 \* a \* c;

if (operand > 0)

{

const auto root1 = (-b + sqrt(operand)) / (2.0 \* a);

const auto root2 = (-b - sqrt(operand)) / (2.0 \* a);

std::cout << "The roots of the equation "

<< a << "x^2 + " << b << "x + " << c << "\n"

<< "are " << std::fixed << std::setprecision(3) << root1 << " and " << root2 << std::endl;

}

else if (operand == 0){

const auto equal\_root = -b / (2.0 \* a);

std::cout << "The root is equal " << std::fixed << std::setprecision(3) << equal\_root << std::endl;

}

else {

const auto equal\_root = -b / (2.0 \* a);

const auto imaginary\_root =sqrt(operand) / (2 \* a);

std::cout << "This is a complex / imaginary root" << std::endl;

std::cout << "Root 1 = " << std::fixed << std::setprecision(3) << equal\_root << " + " << imaginary\_root << " i " << std::endl;

std::cout << "Root 2 = " << std::fixed << std::setprecision(3) << equal\_root << " - " << imaginary\_root << " i " << std::endl;

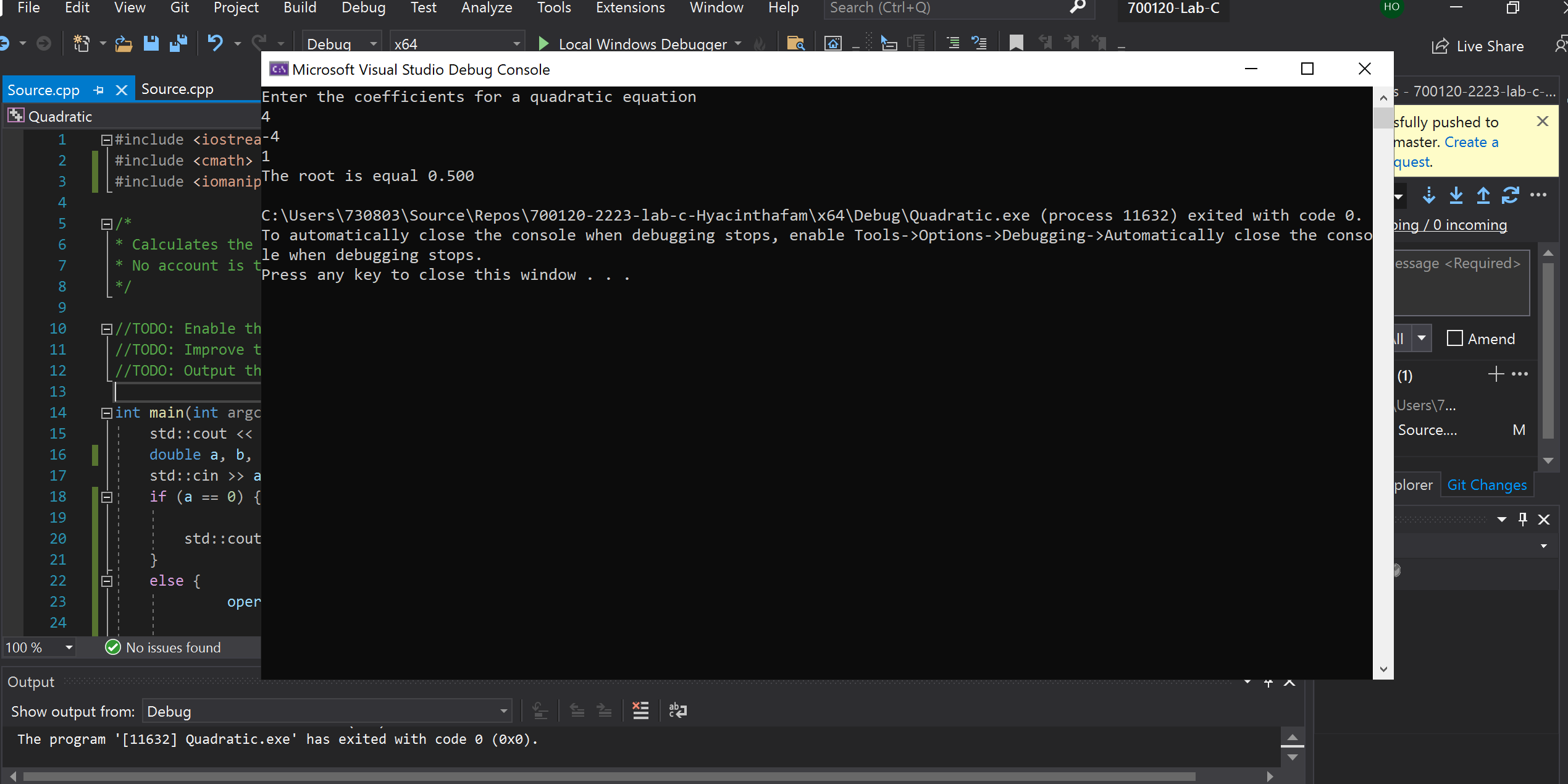
}

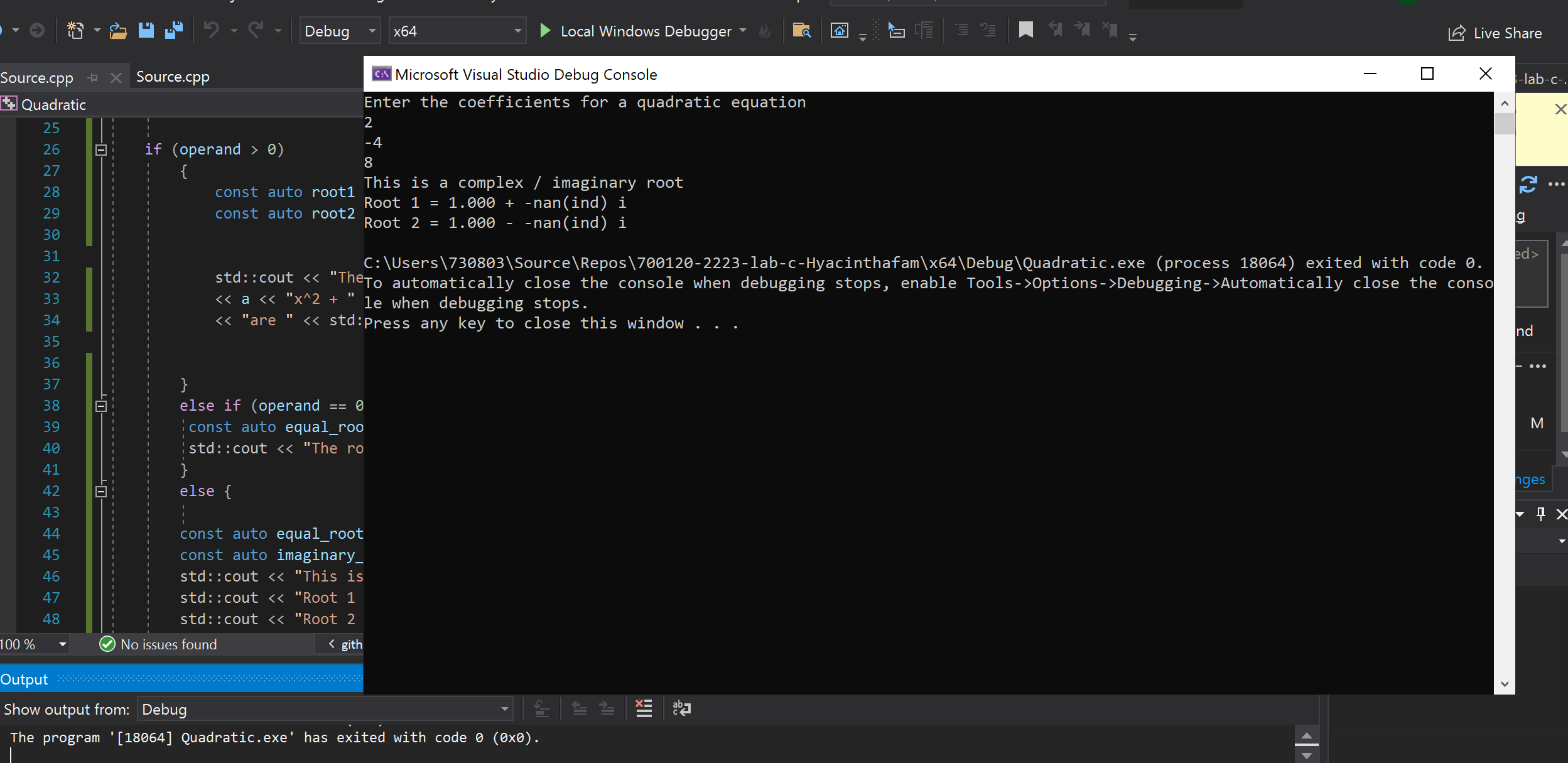
}

return 0;

}

**Sample Output :**

****

****

**Test Data:**

**To test for equal roots 4, -4, 1 was used**

**To test for imaginary / complex roots 2,-4, 8 was used as input values respectively.**

1. Output the roots to only 3 decimal places

## Solution :

Import <iomanip>

<< std::fixed << std::setprecision(3)

Reflections:

The quadratic equation code was modified to handle all quadratics equations using the conditional if else statement, by checking the input value of “a” the compiler will identify the quadratic equation type.

I learnt that to restrict decimals to defined places you have to include the iomanip in the header of the program and subsequently use the fixed and setprecisioon keywords to limit your output to certain number of decimal places.

## Q5. Assembly

Open your solution to the Parsing exercise in Visual Studio. Set a break point in the code and execute the program, so that it halts on the break point.

## Q6. Optimizer

Now that you’re more familiar with assembly code, jump back to your optimized Quadratic code in the Timer project.

Disassembly your solution, this time in Release mode.

Are you able to get a rough understanding of how the optimizer has compiled your C++?

You may find it easier to look at x86 code rather than x64.

Jot down in your log book anything significant that you discover