P

Denali Cornwell & Ryan Larson

UWB CSS 343, Dr. Erika Parsons

Summary

This document contains instructions on how to run the program, the design principals behind the classes/methods, and the testing that was completed to achieve the final state.

Program 4: OOP

Genetic Algorithm for Solving Sudoku

Contents

[Compilation and Running Instructions 2](#_Toc11183423)

[Introduction 2](#_Toc11183424)

[Assumptions 2](#_Toc11183425)

[Classes 3](#_Toc11183426)

[Genetic Algorithm (GeneticAlgorithm.h & GeneticAlgorithm.cpp) 3](#_Toc11183427)

[Puzzle (Puzzle.h) 4](#_Toc11183428)

[Sudoku (Sudoku.h & .cpp) 4](#_Toc11183429)

[Fitness (Fitness.h) 5](#_Toc11183430)

[Sudoku Fitness (SudokuFitness.h & SudokuFitness.cpp) 5](#_Toc11183431)

[Puzzle Factory (PuzzleFactory.h) 5](#_Toc11183432)

[Sudoku Factory (SudokuFactory.h & SusokuFactory.cpp) 5](#_Toc11183433)

[Population (Population.h) 6](#_Toc11183434)

[Sudoku Population (SudokuPopulation.h & SudokuPopulation.cpp) 6](#_Toc11183435)

[Reproduction (Reproduction.h) 6](#_Toc11183436)

[Sudoku Offspring (SudokuOffSpring.h & SudokuOffSpring.cpp) 7](#_Toc11183437)

[Testing 7](#_Toc11183438)

[Outputs 8](#_Toc11183439)

[Valgrind 8](#_Toc11183440)

[cpplint 9](#_Toc11183442)

[Output 10](#_Toc11183443)

[Conclusion 10](#_Toc11183444)

# 

# Compilation and Running Instructions

* **Compile and run with:** g++ \*.cpp; ./a.out
* **Operations Executed On:** 
  + Linux uw1-320-10
  + 4.9.0-8-amd64 #1 SMP Debian 4.9.144-3.1 (2019-02-19) x86\_64

# Introduction

For this program, we were tasked to make a genetic algorithm that attempts to solve sudoku puzzles.

**Class Diagram:**

A screenshot of a cell phone

Description automatically generated

# Assumptions

The program assumes:

* The file name to input as a sudoku is named “default.txt”
* The number of generations do not exceed 7,000 with a population larger than 200
* The first command line argument is the population represented as an integer
* The second command line argument is the number of generations represented as an integer
* A solved sudoku puzzle has 0 collisions and thus a fitness value of 0
* The default.txt file contains 81 integers and any number of special characters, new lines, etc.
* The program is run on either CLion or Linux
* The Sudoku passed in, in “default.txt” is a solvable Sudoku puzzle

# Class Design

This section explains all of the design principals behind each of the classes in our program.

## Genetic Algorithm (GeneticAlgorithm.h & GeneticAlgorithm.cpp)

The genetic algorithm class ties everything together and allows for a generalized approach to solving puzzles with grid styles. This is the most general class in the program and only implements one executable method that will be used in the driver to run the program and get the output.

**Genetic Algorithm Data Flow Diagram:**

A close up of a map

Description automatically generated

A screenshot of a cell phone

Description automatically generatedAs seen in the diagram, we are using the general genetic strategy to complete the sudoku puzzle where in the loop, the program will create a generation based on the previous generation’s survivors, and the number of survivors is equal to 10% of the original population. Upon execution of the genetic algorithm, the puzzle with the lowest fitness value (least number of collisions) is output to the console.

This graph represents the fitness levels in any given generation when compiled using 1900 as the maximum number of generations, and reached as low as 2 for the best fitness level in the final generations. The puzzle approaches fitness levels of 20 and below very quickly and maintains a shallow decline throughout the next thousand generations until finally reaching a fitness level of 2.

It could be stated mathematically that as the number of generations approaches infinity, the fitness levels of the puzzles in the population approaches zero. However, in this case, the fitness value never reaches zero and thus is not solved. This could be as a result of a faulty default puzzle or a consistently reached local minima.

## Puzzle (Puzzle.h)

This is a base class that can be generalized to all puzzles that have a grid type structure. We can say this because we have literally created a grid out of a vector of vectors, that contains pairs. The reason why we chose pairs to store the actual data in our puzzle is because we want to have one layer of actual values, and one layer of Boolean values, this is so that we can keep track of the hints that the program is originally given, so we can make sure not to change those cells later when we mutate a puzzle.

## Sudoku (Sudoku.h & .cpp)

The sudoku class is a child class of puzzle and inherits all of its member functions as well as its “grid” structure. This means that a sudoku has an “is-a” relationship to the puzzle class. <write more here>

The purpose of SudokuFactory was to create puzzles efficiently and without creating unintended duplicates. It handled most of the construction of pointers and was used to instantiate the population to its full size when newGeneration() was called. This was the only class that returned a puzzle pointer but was handled using dynamic\_cast<Sudoku\*>, this could have been changed to return a Sudoku\* if PuzzleFactory was not used to construct it, but it was less code.

## Fitness (Fitness.h)

The fitness class will be able to evaluate the fitness of a puzzle that has a gird. It’s primary function is to return an integer value that will represent a fitness score so that we can see which sudokus are the best in a population. The fitness scores are related as such: the lower the score the better the fitness of a puzzle.

## Sudoku Fitness (SudokuFitness.h & SudokuFitness.cpp)

The Sudoku Fitness class will inherit from the fitness class all of its behaviors and data, it will be a more specific fitness class that will implement evaluative techniques for sudoku puzzles in particular.

The SudokuFitness object was used throughout the program to calculate and return any Sudoku puzzles fitness rating at any given time. Because we designed each of the classes to behave like a family and left data that would regularly be private as protected, it was used with ease.

A SudokuFitness object was designed to be created while passing in a Sudoku\*, and using the pointer it had access to all elements in the grid.

Each row, column, and 3x3 block in the grid were evaluated in the howFit() method. The howFit() method used several nested loops to create vectors containing all elements in any given row, column, and block, and then would use that vector to count the number of collisions. This was done by popping a value off the back of the vector, storing it into a temporary variable as an integer, and then either adding one to the collisions if it is equal to the last element of the vector, or popping off the next item to store.

At the end of the function call it would return the sum of all sums of vectors collisions.

Even though the blocks fitness was evaluated using a quadruple nested loop, the time complexity was still reasonable as the grid was a fixed size.

getFitFromVect() was created to be used for each individual vector and not be used redundantly in the howFit() method. (There were 9 row vectors, 9 column vectors, and 9 block vectors).

## Puzzle Factory (PuzzleFactory.h)

The puzzle factory class is designed to be a “factory” design pattern for our program, it will allow us a convenient way to create many instances of puzzles so that we can populate a population. It uses a puzzle named “original puzzle” so that it makes sure to retain the integrity of the hints as we continue to produce puzzles.

## Sudoku Factory (SudokuFactory.h & SusokuFactory.cpp)

The sudoku factory class is a child class of puzzle and inherits all of its members functions as well as its “original puzzle” data member so that it maintains the integrity of the file read hints of the sudoku.

## Population (Population.h)

The population class is the container for a bunch of puzzles and operates on said container by culling and creating new generations. We decided to use a struct called “individual” which represents a puzzle and its fitness value since we did not hard code in the fitness value of a puzzle into a puzzle. To store these individuals, we made a vector of individuals which is the actual population, and a popSize variable to control the size of the population since vectors are dynamically resizable and don’t adhere to a strict size. This vector of individuals will always be sorted before a culling, it can be sorted because we defined a < operator within Individual that depends on the fitness value. This is important because the cull method will always remove the “back n” elements from the population, leaving the best (smallest fitness value) puzzles to become templates for the next generation.

## Sudoku Population (SudokuPopulation.h & SudokuPopulation.cpp)

The sudoku population class is a child class of the population class, it inherits all of its functions and members. It is a more specific class of the population class and will be able to hold a bunch of individuals that have sudokus puzzles stored in their “puzzle” data.

The Population class stored the number of puzzles intended to be contained in the population vector. This number is received as the first command line argument and is used to determine when to stop inserting puzzles into the population or to determine how many puzzles need to be inserted into the population based off its current size.

The Cull() function was designed to be as simple as possible to reduce the number of areas in which memory leaks may occur. It simply iterated through a designated number of times and popped from the back.

OutputPuzzles() was implemented for testing purposes only. It output every puzzle created, labeled them, and categorized them by their generation. This made it very noticeable when values were incorrect and if the hints were not being handled as expected.

newGeneration() was used to handle the aftermath of Cull(). Whatever Cull() removed from the population, newGeneration would replace by calling Reproduce(), which would insert puzzles into the population vector at the end and then sort it when it is completely filled int. If the generation count was on its first iteration, newGeneration() would use the SudokuFactory class to create new puzzles from the puzzles received in “default.txt”.

BestFitness() and BestIndividual() were easy to implement and great for using to test and display understandable output. BestFitness returns the fitness value associated with the puzzle and BestIndividual would return the puzzle at the beginning of the population vector as it was sorted from least to greatest.

## Reproduction (Reproduction.h)

The reproduction class is used within the population class to mutate and create puzzles based off of the surviving population after a culling. It is primarily used within the newGeneration() method within population because that is the method that creates a new generation in the population after a culling.

## Sudoku Offspring (SudokuOffSpring.h & SudokuOffSpring.cpp)

The Sudoku Offspring class inherits from the reproduction class and allowing for reproduction of sudoku specific puzzles.

This class handled the reproduction of the population after any number of puzzled have been culled based off their fitness levels.

The reproduce function used a custom class called Individual, and what this represented was the pair of Individual puzzles and their associated fitness value. This allowed for the vector of individuals, which was our container for the population, to be stored in a sorted order. The comparison operator needed to be overloaded because std::sort() could not handle self-defined objects and needed a means to compare the pair we were passing in.

This design was advantageous in many ways but had major flaws. It helped because the sorting of the vector made culling the population incredibly easy. Elements could be popped off the back (botXPercentage \* populationSize) number of times because the puzzles with the worst fitness ratings were towards the end of the vector while the bestIndividual() remains in the front (which is also advantageous).

What made vectors flawed in this program was that we wanted to store an Individual class into a vector using push)back or emplace\_back(), which would create copies of the original and store them into the vector. This was problematic because there was unpredictable behavior occurring within the push\_back() calls that would cause memory leaks even if the original was deleted directly after the call to the function. There were also times where it would call the Individual copy constructor up to seven times when it needed only one.

makeOffSpring() was designed to be a very simple mutate function. It simply received a puzzle and iterated through each element and using random numbers would determine with a probability of 5% if it would be randomized between 1 and 9. Each iteration would check if the number was a hint, and skip ahead if it were, otherwise the entire puzzle would be potentially changed and remain in the population it is stored in.

# Testing

During the building phase, we mostly tested for what would compile and run our program as we built it. Once we had all the design down we knew there would be many memory leaks since we instantiated a lot of pointers in our program. As we got to memory leak testing, it became apparent very quickly that we should unit test our code class by class and method by method so that we could trace down any leaks that remained. This stage most definitely took the longest and was the hardest stage of the program.

Unfortunately, we were not able to get rid of all the memory leaks within the program because we ran out of time while testing it. Most of the leaks were coming from the population class that we cleaned up and others coming from the reproduction class. We did the best we could but unfortunately could not solve all the leaks. Our intention was to perform integration testing as well, but the time consumed due to tracking down memory leaks was too extensive.

# 

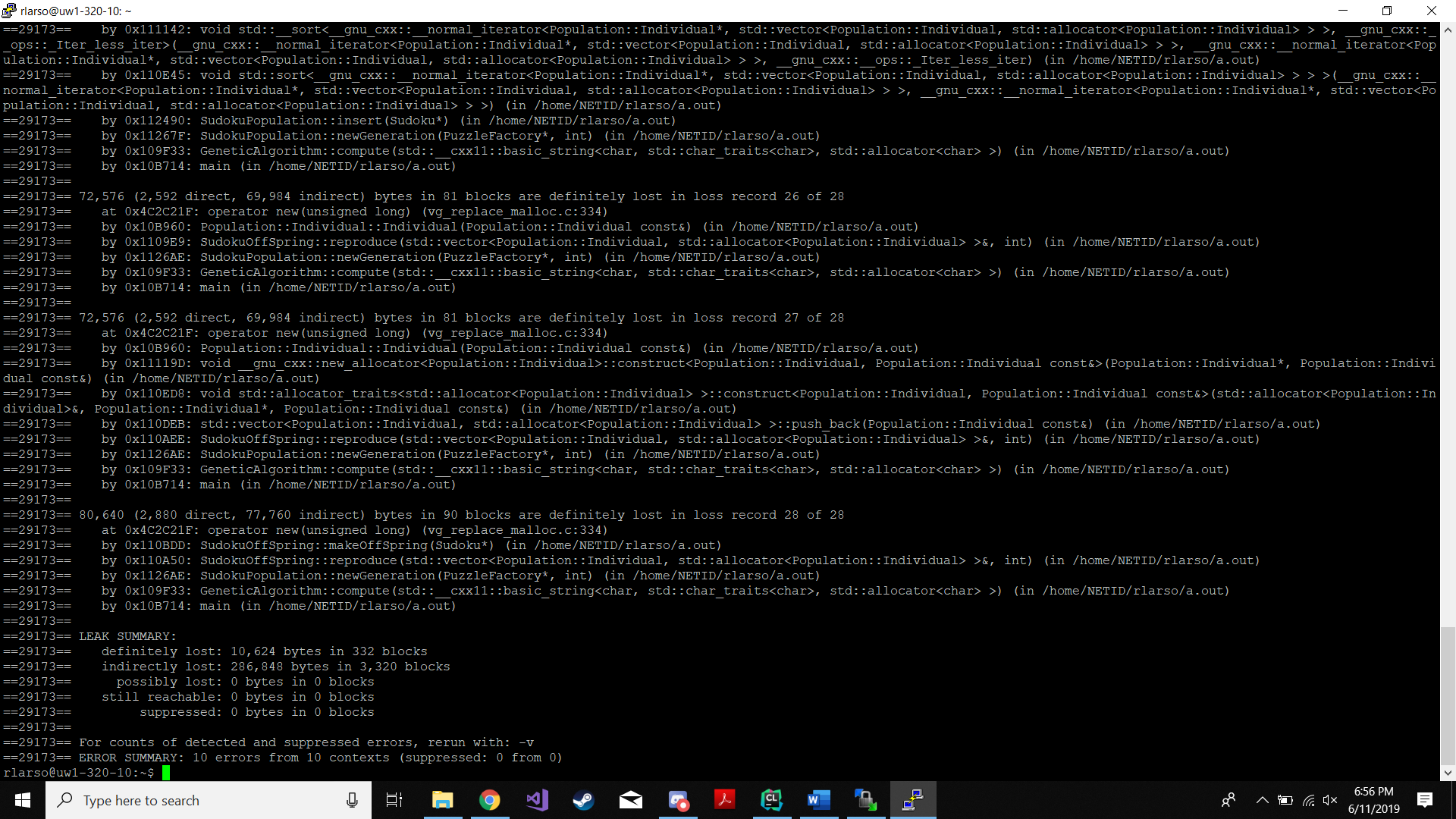
# Outputs

### Valgrind

**Ran with:** valgrind ./a.out

## 

**Ran with:** valgrind –leak-check=full ./a.out



Valgrind Summary:

Memory leaks were the largest problem faced in this program. Because the container for our puzzles was a class called Individuals, and we were using the STL to use vectors, we had to use vector specific functions like push\_back(), emplace\_back(), pop\_back(), and erase(), each of which would cause memory leaks when called in our unit tests.

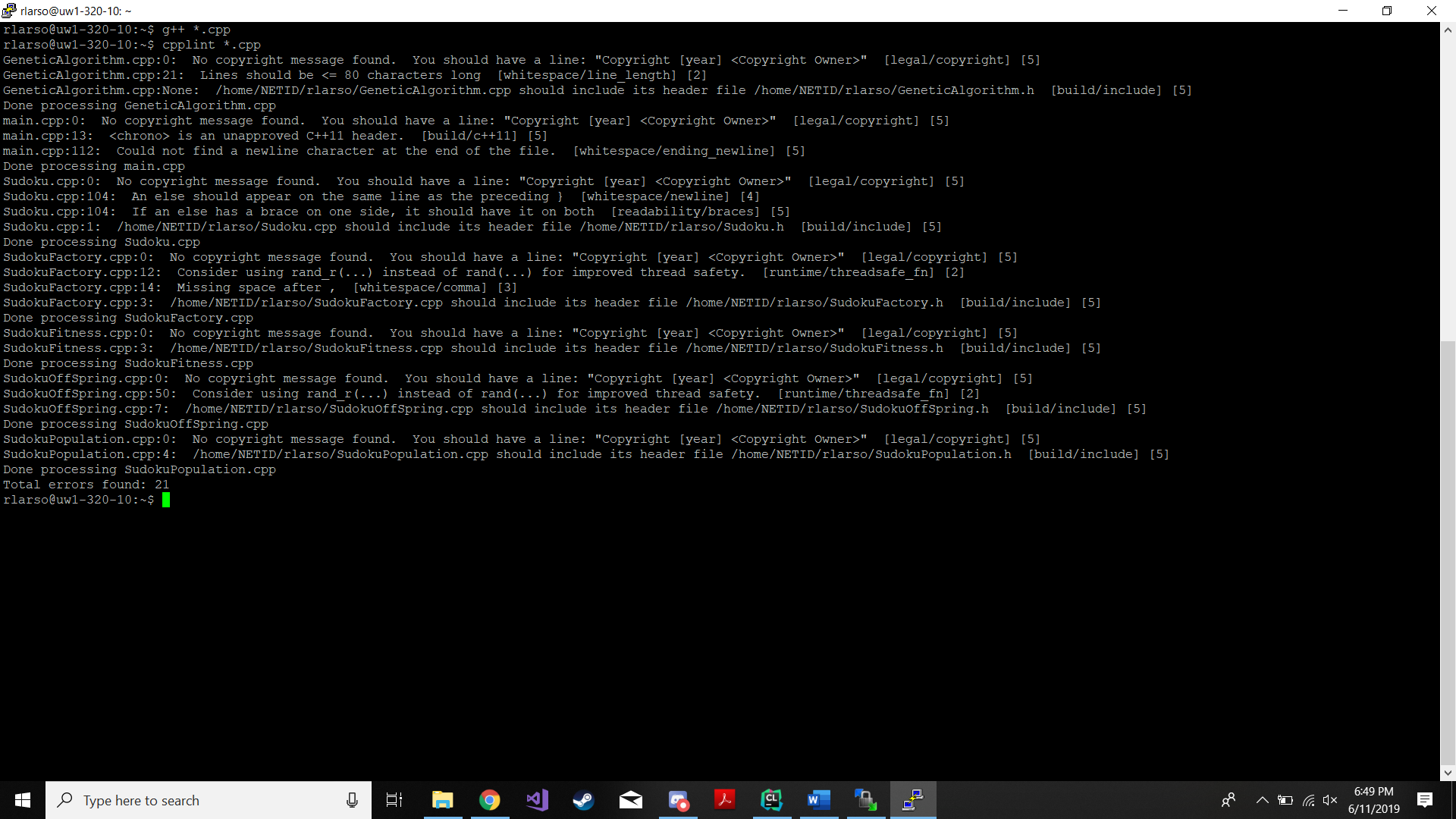
It was found that push\_back(), etc. would create a copy of what is passed in and store the copy, while leaving the original untouched, but even after passing in an Individual object or pointer, and then deleting it, the push\_back() function would for an unknown reason call the copy constructor for the Individual class more than it should have.

Inserting one Individual into the population would cause no memory leaks, a second would cause one puzzle worth of memory lost, and a third insert would cause a memory leak worth 3 puzzles. The pattern continued in an unpredictable manner and resulted in complicated memory leaks that the deconstructors could not handle.

For future programs, the use of a multiset in this situation would be far superior, but with time constraints it would not be likely to make the correct changes to each class’s member functions.

### Cpplint

**Ran with:** cpplint \*.cpp



**Total errors found:** 24

All formatting errors were corrected and all other cpplint errors were considered and determined to be unnecessary for the purposes of this program, or too ugly to fix.

## Linux Program Output

**Ran with:** ./a.out 100 1000

A screenshot of a computer

Description automatically generated

All output of puzzles was intended to match as is shown in the specifications. In the program itself, lines that output entire populations and /or generations remain for testing purposes and took full advantage of the ostream << operator overloading. The execution time was included purely out of interest to see how it compares with higher and lower number of generations and population sizes.

# Conclusion

In conclusion, this program was a challenge, and we had a number of problems with it. We could not button up all the memory leaks in time for the due date but learned a significant amount about how the standard libraries containers work, as well as how to best use them. New methods of testing were realized and put to the best use as possible. Unit testing proved to be the most helpful as far as finding what is functioning and not functioning properly. The practice with debugging and spotting memory leaks may prove to be incredibly powerful when moving on with other programs involving complicated classes and class relationships such as this one. Although it was very time consuming and at times frustrating, overall it was a good experience and we have developed a strong interest in genetic algorithms and their applications in other fields outside of puzzles.