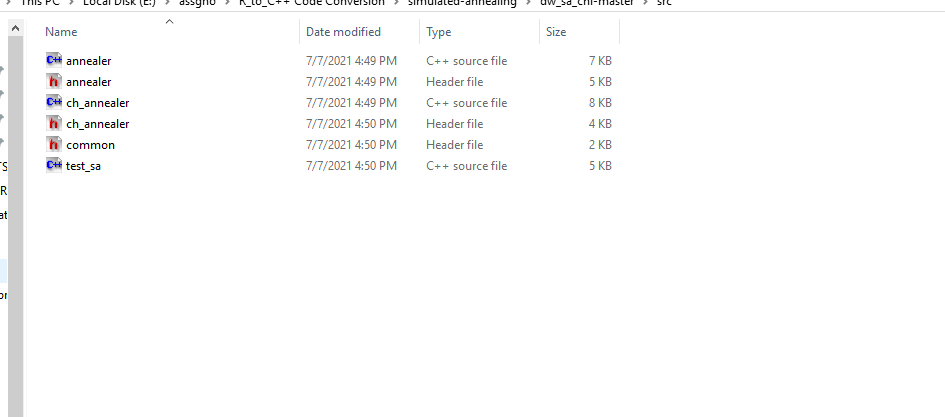
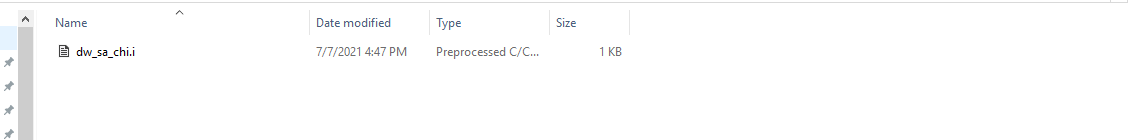
SIMULATED ANNEALING

Program structure

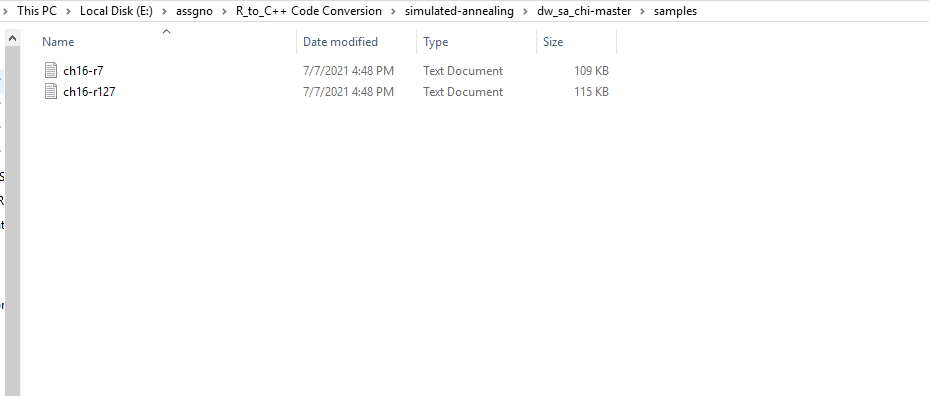
Src files for the program execution



Interface for Execution



Sample files for execution



## 1.1 Structure

The main component of this program is the Genetic class derived from the Algorithm base class; this structure allows me to swap in the prior algorithms with ease (the HillClimber and SimulatedAnnealing classes). When instaniated, this class requires a Problem object, which is a base class representing the interface to the various test functions we were given. Potential solutions to a problem are represented by an Individual class.

**1.2 Individual**

The Individual class’s primary data structure is a C++11 array container, a template similar to a vector, but more efficient and requiring a size (the given dimension of thirty). Thus its genome (potential solution) is an array of thirty parameter types, where parameter is a typedef for double (“parameter” could honestly be “chromosome”, except that it is used elsewhere not as a chromosome but where it should still share the type). It additionally has a fitness parameter, set when created by the Problem class, and updated when mutated by the Algorithm class. The Individual class has a mutate(gene, gene\_i) function whose purpose is to bound-check a mutation operation on any single given gene, and clip it to the domain’s minimum and maximum value as needed. Through the use of templated operator overloads and begin() and end() member functions, an Individual object can be treated like an iterator (of its genome array), compared by fitness such that a greater fitness is defined by its minimization flag (that is, if(minimize) then zero is greater than positive (worse) fitnesses), and arithmetically added by fitness. This makes the implementation of the algorithms very clean.

1.3 The Problem

The Problem class holds the following data: the interval of the domain, the interval of the range (used for normalization), a minimization flag (that is, whether or not the goal is to minimize the problem’s value), the goal itself, and the maximum number of iterations/generations the algorithm should run for that problem. The Problem class provides several member functions: fitness(Individual) which is a virtual function implemented by the derived classes and calculates the fitness score using the definition of the problem’s function, potential() which returns an Individual object instaniated with the data it needs (the domain interval, minimization flag, random potential solution generated by using a uniform\_real\_distribution object, and fitness value for that potential solution), normal(Individual) which normalizes an Individual object’s fitness score onto the interval [0, 1] using the problem’s range (where 1 is the maximum fitness, with minimization taken into account), worst() which finds the worst fitness using an in-place search across iterations number of random potential solutions (used to find an adequate range), and finally represent() which returns a friendly string representation of the problem, including its name and other parameters.