Vote Count

Christian Bradford | DS&A and Advanced Programming | February 9th, 2017

C00223037 | Noel O Hara & Aine Byrne

Contents

[Introduction 2](#_Toc505783428)

[The Problem 2](#_Toc505783429)

[Data Structures 2](#_Toc505783430)

[C++ Code 2](#_Toc505783431)

[main.cpp 2](#_Toc505783432)

[Candidate.h 3](#_Toc505783433)

[Candidate.cpp 3](#_Toc505783434)

[BallotPaper.h 4](#_Toc505783435)

[BallotPaper.cpp 4](#_Toc505783436)

[VoteCounter.h 5](#_Toc505783437)

[VoteCounter.cpp 7](#_Toc505783438)

[Descriptions of Functions 14](#_Toc505783439)

[Candidate.h 14](#_Toc505783440)

[BallotPapers.h 15](#_Toc505783441)

[VoteCounter.h 15](#_Toc505783442)

[Test Data 16](#_Toc505783443)

[Sample Execution (5 Candidates, 17 Voters) 17](#_Toc505783444)

[Sample Execution (3 Candidates, 4 Voters) 19](#_Toc505783445)

# Introduction

This assessment was commissioned by the Institute of Technology Carlow. For the classes of Advanced Programming and Discrete Structures and Algorithms under the supervision of Aine Byrne and Noel O Hara.

# The Problem

Complete a vote-counting application for a simple electoral system based in single-seat constituencies. There are any number of candidates, one which will be elected. Each voter scores the candidates in order of preference 1, 2, 3 etc. on a ballot paper. During the count, each ballot paper is distributed initially to the can­didate who received the highest preference on that paper. Once this is done, candidates are eliminated one by one until only one remains, the last remaining candidate being deemed the winner. At each elimination the candidate with the lowest number of votes is chosen and his votes are re-distributed among the remaining candidates, with each voting paper being awarded to the remaining candidate with the highest preference on that ballot paper. All ties are solved by random choice.

# Data Structures

The data structures that are used, in my solution, to solve the solution are: maps, arrays and, lists. I only used the map data structure inside of the BallotPaper class. The map data structure allows me to easily store all the preferences of all the candidates by using the preference index as the key and the value that matches the key is the Candidate. The array data structure type is only used to store strings in various parts of the code. I use a helper class called CandidateVotes that has two properties: A Candidate and a list of BallotPapers that the Candidate owns.

The list data structure is the most important part of my solution. The list is a doubly linked list. I use the list to hold all the running Candidates and the BallotPapers that are used in the voting process. I also have a list of CandidateVotes that contains all the information about which candidates are left in the race, they haven’t been removed from the race.

# C++ Code

This is the source code for the vote count problem. The code is separated into seven different files. I also will show a sample input file that is used in the program if you wanted to load a file into it.

## main.cpp

//Christian Bradford

//C00223037

//Joint Assignment 2

#include <iostream>

#include "VoteCounter.h"

int main()

{

VoteCounter voteCounter;

voteCounter.Start();

system("pause");

return 0;

}

## Candidate.h

//Christian Bradford

//C00223037

//Joint Assignment 2

#include<string>

#pragma once

class Candidate

{

public:

/\*

Creates a new candidate with the name and party

@param name of the Candidate

@param party of the Candidate

\*/

Candidate(std::string name, std::string party);

Candidate() {}

~Candidate() {}

/\*

@return the name of the Candidate

\*/

std::string const getName();

/\*

@return the party of the Candidate

\*/

std::string getParty();

/\*

@param name sets the name of the Candidate

\*/

void setName(std::string name);

/\*

@param party sets the party of the Candidate

\*/

void setParty(std::string party);

/\*

Allows you to compare the names of the Candidates

\*/

bool operator== (Candidate &d);

/\*

Allows you to compare the names of the Candidates

\*/

bool operator!= (Candidate &d);

private:

std::string \_name;

std::string \_party;

};

## Candidate.cpp

//Christian Bradford

//C00223037

//Joint Assignment 2

#include "Candidate.h"

Candidate::Candidate(std::string name, std::string party)

{

\_name = name;

\_party = party;

}

std::string const Candidate::getName()

{

return \_name;

}

std::string Candidate::getParty()

{

return \_party;

}

void Candidate::setName(std::string name)

{

\_name = name;

}

void Candidate::setParty(std::string party)

{

\_party = party;

}

bool Candidate::operator == (Candidate &d)

{

return \_name == d.getName();

}

bool Candidate::operator != (Candidate &d)

{

return \_name != d.getName();

}

## BallotPaper.h

//Christian Bradford

//C00223037

//Joint Assignment 2

#pragma once

#include "Candidate.h"

#include <map>

class BallotPaper

{

public:

BallotPaper() {}

~BallotPaper() {}

/\*

Sets the Perference of the Choosen Candidate

@param preference is the preference on this ballet for a certian Candidate

@param candidate is the Candidate

\*/

void setPreference(int peference, Candidate candidate);

/\*

@param preference the preference number of the candidate

@return a Candidate of the peference provided

\*/

Candidate getPreference(int preference);

private:

std::map<int, Candidate> \_candidates;

};

## BallotPaper.cpp

//Christian Bradford

//C00223037

//Joint Assignment 2

#include "BallotPaper.h"

void BallotPaper::setPreference(int preference, Candidate candidate)

{

\_candidates[preference] = candidate;

}

Candidate BallotPaper::getPreference(int preference)

{

return \_candidates[preference];

}

## VoteCounter.h

//Christian Bradford

//C00223037

//Joint Assignment 2

#pragma once

#include <iostream>

#include <list>

#include <iterator>

#include <string>

#include <sstream>

#include <stdlib.h>

#include <fstream>

#include "Candidate.h"

#include "BallotPaper.h"

struct CandidateVotes

{

/\*

A pointer to a Candidate

\*/

Candidate c;

/\*

All the ballot Papers that Are there votes

\*/

std::list<BallotPaper> votes;

};

enum ERROR\_CODES

{

CandidatesError = 0,

VotersError = 1,

FileReadError = 2,

};

class VoteCounter

{

public:

VoteCounter() {}

~VoteCounter() {}

/\*

Starts the VoteCounter System by asking if manual process or file process

\*/

void Start();

private:

/\*

Starts the voting process by voting

@param candidates is the list of voteable candidates

\*/

void StartVotingManual(const std::list<Candidate> candidates);

/\*

Asks the user for all the names, and partys of the Candidates

@return a list of Candidates that can be voted for

\*/

std::list<Candidate> InitManualProcess();

/\*

Print out All the Candidates

@param candidates is a list of candidates that you would like to be printed out

\*/

void PrintOutAllCandidates(std::list<Candidate> candidates);

/\*

Algorthim to Figure Out who won based on a list of Ballot Papers

@param ballotPapers is a list of Ballot Papers that contain the voting preferences

@param candidates is the list of voteable candidates

\*/

void FigureOutWhoWon(std::list<BallotPaper> ballotPapers, const std::list<Candidate> &candidates);

/\*

This will take the list of Ballot papers from the Loser Candidate

and put the papers into the next candidate based on preference

@param loserCandidate is the Candidate who just lost

@param votes is the list of still running Candidates

\*/

void TransferVotes(CandidateVotes loserCandidate, std::list<CandidateVotes> &votes);

/\*

This counts the total number of votes for each candidate and returns a list

@param candidates is the list of voteable candidates

@param ballotPapers is a list of Ballot Papers that contain the voting preferences

@return a list of CandidateVotes that contain the number of votes for each candidate

\*/

const std::list<CandidateVotes> CountVotes(const std::list<Candidate> &candidates, const std::list<BallotPaper> &ballotPapers);

/\*

This takes in a list of the current running Candidates and thier votes and returns a list of all the lowest voted Candidates

@param votes is the list of still running candidates

@return a list of CandidateVotes that contain the lowest number of votes for each candidate that tied

\*/

std::list<CandidateVotes> FindAllTheLowestVotes(const std::list<CandidateVotes>& votes);

/\*

Prints out the Voters and their Preferences

@param ballotPapers is a list of Ballot Papers that you have left

@param numCandidates is the number of total Candidates

\*/

void PrintOutVotersAndTheirPreferences(std::list<BallotPaper> ballotPapers, int numCandidates);

/\*

This prints out all the candidates that tied with the lowest vote

@param votes is the list of still running candidates with the lowest votes

\*/

void PrintOutTiedCandidates(const std::list<CandidateVotes>& votes);

/\*

This returns the LoserCandidate that is from the lowest votes

@param votes is the list of still running candidates with the lowest votes

@return a CandidateVote object that is the lost Candidate

\*/

CandidateVotes FindTheLoserCandidate(std::list<CandidateVotes>& votes);

/\*

Prints out the Candidates and their Votes

@param votes is the current running Candidates and thier votes

@param round is the total round that the voting process is in

\*/

void PrintOutCandidatesAndThierVotes(const std::list<CandidateVotes> &votes, int round);

/\*

This takes in a list of CandidateVotes and returns the list in Descending order

@param votes is a list of the still running Candidates

@return a list of the same CandidateVotes list just in descending order

\*/

std::list<CandidateVotes> VoteCounter::SortDesending(const std::list<CandidateVotes> &totalVotes);

/\*

This checks to see if the user entered in a real number

@return a int processed by the user

\*/

int CheckForValidNumberInput();

/\*

This asks if the user is done something

@param message is the yes or no question you would like to ask

@return true if the answer is yes and false if the answer is no

\*/

bool AskForInput(std::string message);

/\*

Validates the input of the user for 'y' or 'n'

@return a validated 'y' or 'n'

\*/

char CheckForValidQuitInput();

/\*

Starts the voting process from a file

@param fileName is the relative path to the file

\*/

void StartVotingFile(std::string fileName);

/\*

This takes in a list of strings and coverts them into a list of Candidates

@param candidatesString is from the file

@return a list of Candidates converted from strings

\*/

std::list<Candidate> ConvertToCandidates(std::list<std::string> candidatesString);

/\*

This takes in a list of strings and coverts them into a list of BallotPapers

@param votesString is from the file

@param candidates is a list of voteable candidates

@return a list of BallotPapers converted from strings

\*/

std::list<BallotPaper> ConvertToBallotPapers(std::list<std::string> votesString, const std::list<Candidate> &candidates);

};

## VoteCounter.cpp

//Christian Bradford

//C00223037

//Joint Assignment 2

#include "VoteCounter.h"

void VoteCounter::Start()

{

char answer = 0;

while (true)

{

std::cout << "Would you like to do a manual process or load a file? (m | l) ";

std::cin >> answer;

if (answer == 'l')

{

std::string filename = "../Data/Vote\_Results.txt";

StartVotingFile(filename);

break;

}

else if (answer == 'm')

{

std::list<Candidate> candidates = InitManualProcess();

StartVotingManual(candidates);

break;

}

}

}

void VoteCounter::StartVotingManual(const std::list<Candidate> candidates)

{

std::list<BallotPaper> ballotPapers;

std::list<Candidate> currentCandidates;

int numVoters = 0;

int currentPick;

bool answer = false;

std::cout << std::endl << "Lets start the voting process!" << std::endl;

while (answer == false)

{

BallotPaper \*currentBallotPaper = new BallotPaper();

currentPick = 0;

++numVoters;

currentCandidates = candidates;

while (currentCandidates.size() >= 1)

{

PrintOutAllCandidates(currentCandidates);

currentPick++;

int index = 0;

while (index < 1 || index > currentCandidates.size())

{

std::cout << "Voter #: " << numVoters << " Current Pick: " << currentPick << " (Pick The Candidate Number)? ";

index = CheckForValidNumberInput();

}

std::list<Candidate>::iterator it1 = currentCandidates.begin();

std::advance(it1, --index);

currentBallotPaper->setPreference(currentPick, \*it1);

currentCandidates.erase(it1);

}

ballotPapers.push\_back(\*currentBallotPaper);

delete currentBallotPaper;

std::cout << std::endl;

answer = AskForInput("Would you like to add another Voter?");

}

FigureOutWhoWon(ballotPapers, candidates);

}

void VoteCounter::FigureOutWhoWon(const std::list<BallotPaper> ballotPapers, const std::list<Candidate> &candidates)

{

int round = 0;

std::list<CandidateVotes> totalVotes;

//Candidates and their votes

totalVotes = CountVotes(candidates, ballotPapers);

//Print Progress

PrintOutVotersAndTheirPreferences(ballotPapers, candidates.size());

while (totalVotes.size() > 1)

{

//Sort out the list in ascending order

totalVotes = SortDesending(totalVotes);

//Print out the Candidates names and the number of votes that they have

PrintOutCandidatesAndThierVotes(totalVotes, round);

//Find The candidate(s) in last place

std::list<CandidateVotes> lowestVotes = FindAllTheLowestVotes(totalVotes);

CandidateVotes loserCandidate;

if (lowestVotes.size() > 1)

{

PrintOutTiedCandidates(lowestVotes);

loserCandidate = FindTheLoserCandidate(lowestVotes);

}

else

loserCandidate = \*lowestVotes.cbegin();

//Print out the lost Candidate

std::cout << "Candidate " << loserCandidate.c.getName() << " has lost the race... " << std::endl;

//Find out where the votes goes too

TransferVotes(loserCandidate, totalVotes);

//Get rid of the loser Candidate from the list of votes

std::list<CandidateVotes> newList;

for (auto cv : totalVotes)

{

if (cv.c != loserCandidate.c)

newList.push\_back(cv);

}

totalVotes = newList;

round++;

}

PrintOutCandidatesAndThierVotes(totalVotes, round);

std::cout << "The winner is Candidate: " << (totalVotes.begin())->c.getName() << std::endl;

}

void VoteCounter::TransferVotes(CandidateVotes loserCandidate, std::list<CandidateVotes> &totalVotes)

{

std::cout << "The vote(s) that went to Candidate " << loserCandidate.c.getName() << " is going to:" << std::endl;

if (loserCandidate.votes.size() < 1)

std::cout << "There was no votes for Candidate " << loserCandidate.c.getName() << std::endl;

else

{

std::list<BallotPaper> transferBallotPapers;

for (auto bp : loserCandidate.votes)

{

//You can take the round number and add one to get to the round preference since round will always be less then total starting Candidates

Candidate tmp;

int prefence = 2; //Start off with the second choice

bool exists = false;

while (exists == false)

{

tmp = bp.getPreference(prefence);

for (auto cv : totalVotes)

{

if (cv.c == tmp && cv.c != loserCandidate.c)

{

exists = true;

break;

}

}

prefence++;

}

std::cout << "Candidate: " << tmp.getName() << std::endl;

for (std::list<CandidateVotes>::iterator it = totalVotes.begin(); it != totalVotes.end(); ++it)

{

if (it->c == tmp)

{

it->votes.push\_back(bp);

break;

}

}

}

}

}

const std::list<CandidateVotes> VoteCounter::CountVotes(const std::list<Candidate> &candidates, const std::list<BallotPaper> &ballotPapers)

{

std::list<CandidateVotes> totalVotes;

//Set up the list of the CandidateVote Structure

for (std::list<Candidate>::const\_iterator it = candidates.cbegin(); it != candidates.cend(); ++it)

{

CandidateVotes tmp;

Candidate c = Candidate(\*it);

tmp.c = c;

totalVotes.push\_back(tmp);

}

//Count the votes and add it to the correct candidate

for (auto bp : ballotPapers)

{

Candidate first = bp.getPreference(1);

for (std::list<CandidateVotes>::iterator it = totalVotes.begin(); it != totalVotes.end(); ++it)

{

if (it->c.getName() == first.getName())

{

it->votes.push\_back(bp);

break;

}

}

}

return totalVotes;

}

std::list<CandidateVotes> VoteCounter::FindAllTheLowestVotes(const std::list<CandidateVotes> &votes)

{

//The list can have mulitple ties so add all the lowest counts to another list

std::list<CandidateVotes> lowestVotes;

int lowestVoteCount = -1;

//Find the lowestVoteCount

for (std::list<CandidateVotes>::const\_iterator it = votes.cbegin(); it != votes.cend(); ++it)

{

if (it->votes.size() < lowestVoteCount)

lowestVoteCount = it->votes.size();

}

//Add to the return list if it matches the lowest vote count

for (std::list<CandidateVotes>::const\_iterator it = votes.cbegin(); it != votes.cend(); ++it)

{

if (it->votes.size() == lowestVoteCount)

lowestVotes.push\_back(\*it);

}

return lowestVotes;

}

void VoteCounter::PrintOutTiedCandidates(const std::list<CandidateVotes> &votes)

{

std::string candidateNames;

int stopIndex = votes.size() - 1;

int currentIndex = 0;

for (auto cv : votes)

{

candidateNames += cv.c.getName();

if (currentIndex < stopIndex)

candidateNames += ", ";

currentIndex++;

}

std::cout << "These Candidates are tied: " << candidateNames << ". Randomly selecting which one is kicked out of the race...." << std::endl;

}

CandidateVotes VoteCounter::FindTheLoserCandidate(std::list<CandidateVotes> &votes)

{

int listIndex = (rand() % votes.size()); // rand % starts at one and the indexing starts at one

std::list<CandidateVotes>::const\_iterator it1 = votes.cbegin();

std::advance(it1, listIndex);

return \*it1;

}

void VoteCounter::PrintOutCandidatesAndThierVotes(const std::list<CandidateVotes> &votes, int round)

{

std::cout << std::endl << "Printing out round # " << round << std::endl;

for (auto v : votes)

std::cout << "Candidate " << v.c.getName() << " has " << v.votes.size() << " votes" << std::endl;

}

void VoteCounter::PrintOutVotersAndTheirPreferences(std::list<BallotPaper> ballotPapers, int numCandidates)

{

std::cout << "Printing the voters and their preference of highest to lowest." << std::endl;

int numVoter = 0;

for (auto bp : ballotPapers)

{

std::cout << "Voter #" << ++numVoter << ": ";

for (int i = 0; i < numCandidates; i++)

{

std::cout << bp.getPreference(i + 1).getName();

if (i + 1 < numCandidates)

std::cout << ", ";

}

std::cout << std::endl;

}

std::cout << std::endl;

}

std::list<Candidate> VoteCounter::InitManualProcess()

{

int count = 0;

bool answer = false;

std::list<Candidate> candidates;

while (answer == false)

{

std::string name;

std::string party;

std::cout << "What is the name of Candidate #" << ++count << "? ";

std::cin >> name;

std::cout << "What is the party of Candidate #" << count << "? ";

std::cin >> party;

Candidate \*tmp = new Candidate(name, party);

candidates.push\_back(\*tmp);

std::cout << std::endl;

answer = AskForInput("Would you like to add another Candidate?");

}

return candidates;

}

void VoteCounter::PrintOutAllCandidates(std::list<Candidate> candidates)

{

int count = 0;

for (auto c : candidates)

std::cout << "Candidate # " << ++count << " Name: " << c.getName() << " Party: " << c.getParty() << std::endl;

}

std::list<CandidateVotes> VoteCounter::SortDesending(const std::list<CandidateVotes> &totalVotes)

{

std::list<CandidateVotes> returnList;

bool added = false;

for (auto tv : totalVotes)

{

bool added = false;

if (returnList.size() == 0)

{

returnList.push\_front(tv);

continue;

}

std::list<CandidateVotes>::const\_iterator it1 = returnList.cbegin();

while (it1 != returnList.cend())

{

if (it1->votes.size() <= tv.votes.size())

{

returnList.insert(it1, tv);

added = true;

break;

}

std::advance(it1, 1);

}

if (added == false) //means thats the tv object is the largest in the list so put it at the end

returnList.push\_back(tv);

}

return returnList;

}

int VoteCounter::CheckForValidNumberInput()

{

int returnValue = 0;

while (!(std::cin >> returnValue))

{

std::cin.clear();

std::cin.ignore(999, '\n');

std::cout << "Please type in a number!!" << std::endl;

std::cout << "Type in a number: ";

}

return returnValue;

}

bool VoteCounter::AskForInput(std::string message)

{

std::cout << std::endl;

std::cout << message << " (y|n): ";

char returnValue = CheckForValidQuitInput();

std::cin.clear();

std::cin.ignore(999, '\n');

std::cout << std::endl;

if (returnValue == 'y')

return false;

return true;

}

char VoteCounter::CheckForValidQuitInput()

{

char returnValue = 0;

bool validInput = false;

bool failedFlag = false;

while (!validInput)

{

std::cin.clear();

std::cin.sync();

if (failedFlag)

std::cout << "Type in a answer: ";

std::cin >> returnValue;

if (returnValue == 'y' || returnValue == 'n')

validInput = true;

else

{

returnValue = 0;

failedFlag = true;

}

}

return returnValue;

}

void VoteCounter::StartVotingFile(std::string fileName)

{

char ignoreCharacter = '#';

std::string seperateCharacter = "";

bool isCandidates = true;

std::string line;

std::list<std::string> listOfInputs = {};

std::list<std::string> candidatesString = {};

std::list<std::string> votersString = {};

std::ifstream myfile(fileName);

try

{

if (myfile.is\_open())

{

while (getline(myfile, line))

{

if (line[0] != ignoreCharacter)

{

if (line == seperateCharacter)

{

if (isCandidates)

{

isCandidates = false;

candidatesString = listOfInputs;

listOfInputs = {};

}

}

else

listOfInputs.push\_back(line);

}

}

myfile.close();

votersString = listOfInputs;

}

else

{

throw FileReadError;

}

std::list<Candidate> candidates = ConvertToCandidates(candidatesString);

if (candidates.size() <= 0)

throw CandidatesError;

std::list<BallotPaper> voters = ConvertToBallotPapers(votersString, candidates);

if (voters.size() <= 0)

throw VotersError;

FigureOutWhoWon(voters, candidates);

}

catch (ERROR\_CODES error)

{

switch (error)

{

case CandidatesError:

std::cout << "The file is not set up properly. Please fix the Candidates and try again!" << std::endl;

break;

case VotersError:

std::cout << "The file is not set up properly. Please fix the Votes and try again!" << std::endl;

break;

case FileReadError:

std::cout << "Can not read the file. Please fix it and try again!" << std::endl;

break;

default:

break;

}

return;

}

}

std::list<Candidate> VoteCounter::ConvertToCandidates(std::list<std::string> candidatesString)

{

/\*

Each line will be set as NAME,PARTY for the Candidate

\*/

std::list<Candidate> returnList;

bool isCreated = false;

Candidate tmp;

for (auto line : candidatesString)

{

std::stringstream ss(line);

while (ss.good())

{

std::string substr;

getline(ss, substr, ',');

if (isCreated == false)

{

tmp = {};

tmp.setName(substr);

isCreated = true;

}

else

{

tmp.setParty(substr);

returnList.push\_back(tmp);

isCreated = false;

}

}

}

return returnList;

}

std::list<BallotPaper> VoteCounter::ConvertToBallotPapers(std::list<std::string> votesString, const std::list<Candidate> &candidates)

{

/\*

Each line will be set as firstPrefence,SecondPrefenece,ThirdPrefenece

Choosing these numbers are based on the index of the candidates listed in the file

\*/

std::list<BallotPaper> returnList = {};

try

{

std::list<Candidate>::const\_iterator it1;

for (auto line : votesString)

{

std::stringstream ss(line);

int i;

int preference = 0;

BallotPaper tmp;

while (ss >> i)

{

it1 = candidates.cbegin();

std::advance(it1, i - 1);

tmp.setPreference(++preference, \*it1);

if (ss.peek() == ',')

ss.ignore();

}

returnList.push\_back(tmp);

}

return returnList;

}

catch (const std::exception&)

{

return returnList;

}

}

# Descriptions of Functions

## Candidate.h

**Candidate():** this the default constructor.

**Candidate(std::string name, std::string party):** This is just a constructor that allows you to set the name and the party of the candidate when you create the object.

**~Candidate():** This the deconstructor.

**std::string const GetName():** This returns the name of the Candidate.

**std::string GetParty():** This returns the party of the Candidate.

**void SetName(std::string name):** This takes in a parameter and replaces the name of the Candidate with the parameter.

**void SetParty(std::string party):** This takes in a parameter and replaces the party of the Candidate with the parameter.

**bool operator== (Candidate &d):** This overloads the ‘==’ operator. This allows comparison between two Candidate objects.

**bool operator!= (Candidate &d):** This overloads the ‘!=’ operator. This allows comparison between two Candidate objects.

## BallotPapers.h

**BallotPaper():** This is the default constructor.

**~BallotPaper():** This is the deconstructor.

**void setPreference(int preference, Candidate candidate):** This stores the Candidate in a map data structure using preference as the key and candidate as the value stored along with the key.

**Candidate getPreference(int preference):** This returns the candidate with the chosen preference on this BallotPaper.

## VoteCounter.h

**VoteCounter():** This is the default constructor.

**~VoteCounter():** This is the deconstructor.

**void Start():** This starts the Vote Counter Program.

**void StartVotingManual(const std::list<Candidate> candidates):** Starts the manual process of the vote counter problem.

**std::list<Candidate> InitManualProcess():** This function asks the user for all the candidate details.

**void PrintOutAllCandidates(std::list<Candidate> candidates):** This prints out all the candidate’s details to the screen.

**void FigureOutWhoWon(std::list<BallotPaper> ballotPapers, const std::list<Candidate> &candidates):** This is the function that determines who wins the voting process.

**void TransferVotes(CandidateVotes loserCandidate, std::list<CandidateVotes> &votes):** This function transfers all the loserCandidate BallotPapers that are associated with it and assigns them to another candidate based on the next candidate preference.

**const std::list<CandidateVotes> CountVotes(const std::list<Candidate> &candidates, const std::list<BallotPaper> &ballotPapers):** This takes all the candidates and ballotPapers lists and creates a CandidateVote object, with a single candidate and all the ballotPapers that are associated with the candidate, and stores it in a returnable list.

**std::list<CandidateVotes> FindAllTheLowestVotes(const std::list<CandidateVotes>& votes):** This functions looks at a list of CandidateVote objects and determines which of those objects have the lowest vote count and returns a list of all the CandidateVote objects with those lowest votes.

**void PrintOutVotersAndTheirPreferences(std::list<BallotPaper> ballotPapers, int numCandidates):** This prints out the ballotPapers and the preference order of the Candidates to the screen.

**void PrintOutTiedCandidates(const std::list<CandidateVotes>& votes):** This prints out the tied candidates to the screen.

**CandidateVotes FindTheLoserCandidate(std::list<CandidateVotes>& votes):** This takes in a list of CandidateVote objects and determines which one of them loses.

**void PrintOutCandidatesAndThierVotes(const std::list<CandidateVotes> &votes, int round):** This prints out the number of votes each candidate has to the screen.

**std::list<CandidateVotes> SortDesending(const std::list<CandidateVotes> &totalVotes):** This takes in a list and returns a sorted list based on the number of BallotPapers each candidate has associated with them.

**int CheckForValidNumberInput():** This function asks a user for a valid number input and returns a validated number input.

**bool AskForInput(std::string message):** This function asks a yes or no question and returns a validated Boolean answer.

**char CheckForValidQuitInput():** This function asks a yes or no question and returns a validated ‘y’ or ‘n’ char.

**void StartVotingFile(std::string fileName):** This starts the vote counting program based on a loaded file.

**std::list<Candidate> ConvertToCandidates(std::list<std::string> candidatesString):** This takes a list of string that is from a file and converts them into Candidate objects.

**std::list<BallotPaper> ConvertToBallotPapers(std::list<std::string> votesString, const std::list<Candidate> &candidates):** This takes a list of string that is from a file and converts them into BallotPaper objects.

# Test Data

For the test data I have taken only two different files that are set up as the following:

The first file has five different Candidates and seventeen different voters to make the BallotPapers. The file is set up as:

#Candidates (name,party)

one,one

two,two

three,three

four,four

five,five

#Voters (firstPreference,secondPreference,thirdPreference)

1,4,3,2,5

2,3,4,5,1

1,4,3,5,2

2,3,4,5,1

3,1,4,5,2

1,5,4,3,2

2,5,4,3,1

3,2,4,5,1

4,2,1,5,3

1,3,4,5,2

2,4,1,5,3

3,1,2,4,5

4,3,1,5,2

5,2,1,3,4

5,2,3,1,4

4,1,2,3,5

5,4,3,2,1

This data is set up to simulate a few different ties and BallotPapers having to go to the third or fourth preference for the Candidates. With this data the whole race will finish in 5 rounds and Candidate two will always win.

The second file is much simpler has it only has three candidates and four voters to make the BallotPapers. The file is set up as:

#Candidates ( name,party)

one,one

two,two

three,three

#Voters (firstPreference,secondPreference,thirdPreference)

1,2,3

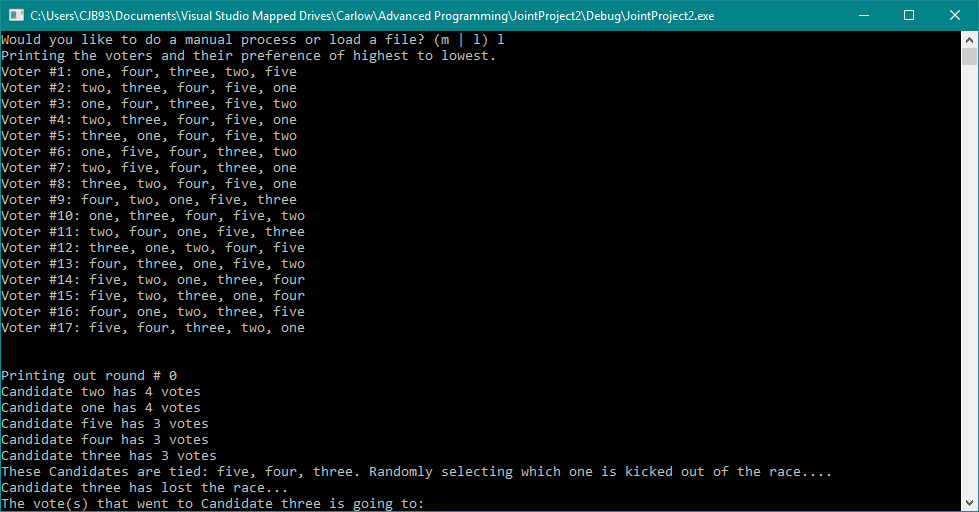
2,3,1

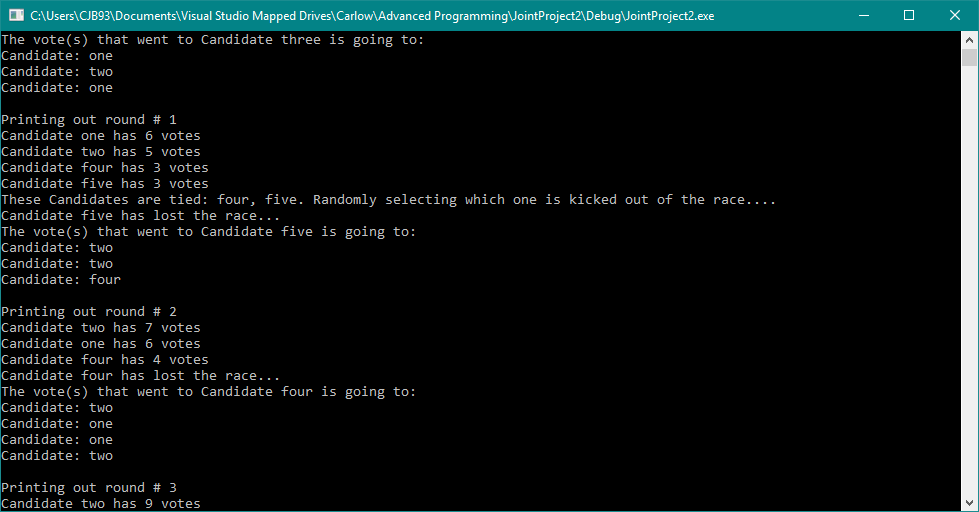
3,1,2

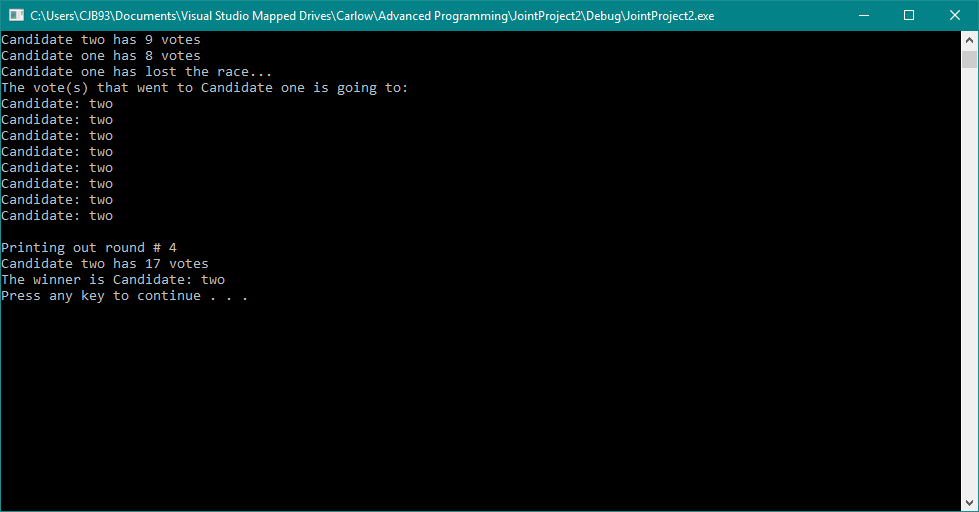
1,2,3

This data was used to determine of how the randomness of determining who lost based on a tie can affect the outcome of the race. The first round Candidate two and three tie and if Candidate two loses in the first round, then in next round there’s another tie between Candidate one and three. The winner if Candidate two is the first one out is a 50/50 split between Candidate one and three. Now if Candidate three loses in the first round, instead of Candidate two, then Candidate one wins every time.

# Sample Execution (5 Candidates, 17 Voters)







# Sample Execution (3 Candidates, 4 Voters)

