**Sabastian Fasano**

**CS-300**

**Jan. 12, 2022**

**Module 2 Vector Sorting Reflections**

The code is broken down into the following functions/structs:

* strToDouble
  + Used to convert the CSV file data into useable value
* Bid
  + Struct containing the data
  + Used with the vector that will be sorted
* displayBid
  + Used to send the values contained in the vector to the console
* getBid
  + Unused
* loadBids
  + Function used to read in the csv data
  + Can read the csv path in from arguments or use a default path
  + Add parts of the data into the Bid structure and then adds that Bid to the unsorted vector
* partition
  + Function that calculates the partition for the quick sort of function
  + *partition* tries to find the elements in half of the of the vector that are out of order and then swap them
* quickSort
  + *quicksort* is a recursive function that drives the sorting via the partition function
* selectionSort
  + *selectionSort* is a function that uses the selection sort algorithm
  + there are 2 loops that scan for vector elements out of order and then swaps the elements
* main
  + *main* is the primary driver for the application
  + *main* has a menu to load the data, view the data, sort via *quickSort* or *selectionSort* algorithms and then exit the application
  + *main* also reports the timing each algorithm takes to perform the sort using the *time.h* library

The code was straight forward, especially since the parser was delivered to me. The only issue I faced was with Visual Studio throwing errors on *cin* and *cout* saying they were ambiguous. I have no idea why as I had the std namespace designated but I solved them by defining it as *std::cout* and the errors went away.

Pseudocode:

**Main** Function()

**Read** cmd arguments

**Store** argument as CSV file path

**If** no cmd arguments load default CSV file path

**Loop** while choice is not equal to ‘9’

**Output** menu

**Get** user input; Store in choice

**Validate** user input

**If** choice is not 1-4 or 9 throw an error

**If** choice equals ‘1’

**Start** the clock and **store** in ticks

**Call** loadBids and store CSV data in struct bids

**Output** number of records in the CSV file

**Stop** the clock

**Output** the elapsed time needed to read in the CSV file

**If** choice equals ‘2’

**Loop** through all the records in vector bids

**Call** displayBids()

**If** choice equals ‘3’

**Start** the clock and store in ticks

**Call** selectionSort passing bids

**Stop** the clock

**Output** the elapsed time needed to sort the vector

**If** choice equals ‘4’

**Start** the clock and store in ticks

**Call** quicksort() passing bids

**Stop** the clock

**Output** the elapsed time needed to sort the vector

**If** Choice equals ‘9’

**Exit** the application

**Output** ‘Good bye’

**End**

**selectionSort()**

**Get** vector to sort

**Loop** from smallest index to penultimate index of vector

**Set** *indexSmallest* to the current index position

**Loop** from the current index of outer loop through vector

**Compare** vector element at loop iterator to vector element at *indexSmallest*

**If** vector element at loop iterator is less than vector element at *indexSmallest* make *indexSmallest* equal to loop iterator

**Swap** the vector element at *indexSmallest* with the vector element at the outer loop’s position

**End**

**quicksort()**

**Get** vector to sort, lowest index of vector and highest index of vector

**If** lowest index if greater than or equal to highest index return nothing

**Call** partition() function

**Set** *lowEndIndex* equal to the value returned by the partition function

Recursively **call** quicksort passing the vector, lowest index, and *lowEndIndex* (from above)

Recursively **call** quicksort passing the vector, *lowEndIndex* (from above) plus one, and highest index

**End**

partition()

**Get** the vector to partition, the lowest index and the highest index

Determine the vector element at the midpoint between the lowest and highest index

**Set** pivot equal to this vector element

**Loop** until the lowest index is greater than or equal to the highest index

**Loop** through the vector from lowest index until a vector element larger than the pivot is found

**Overwrite** lowest index with this element’s position

**Loop** through the vector from lowest index until a vector element smaller than the pivot is found

**Overwrite** highest index with this element’s position

**Swap** the vector elements at the new highest and lowest index

**Overwrite** the lowest index by incrementing it one

**Overwrite** the highest index by decrementing it one

**Return** the highest index

**End**

//============================================================================

// Name : VectorSorting.cpp

// Author : Sabastian Fasano; CS300; Jan. 18, 2025

// Version : 1.0

// Copyright : Copyright © 2021 SNHU COCE

// Description : Vector Sorting Algorithms

//============================================================================

#include <algorithm>

#include <iostream>

#include <time.h>

#include <Windows.h>

#include "CSVparser.hpp"

using namespace std;

//============================================================================

// Global definitions visible to all methods and classes

//============================================================================

// forward declarations

double strToDouble(string str, char ch);

// define a structure to hold bid information

struct Bid {

string bidId; // unique identifier

string title;

string fund;

double amount;

Bid() {

amount = 0.0;

}

};

//============================================================================

// Static methods used for testing

//============================================================================

/\*\*

\* Display the bid information to the console (std::out)

\*

\* @param bid struct containing the bid info

\*/

void displayBid(Bid bid) {

std::cout << bid.bidId << ": " << bid.title << " | " << bid.amount << " | "

<< bid.fund << endl;

return;

}

/\*\*

\* Prompt user for bid information using console (std::in)

\*

\* @return Bid struct containing the bid info

\*/

Bid getBid() {

Bid bid;

std::cout << "Enter Id: ";

cin.ignore();

getline(cin, bid.bidId);

std::cout << "Enter title: ";

getline(cin, bid.title);

std::cout << "Enter fund: ";

cin >> bid.fund;

std::cout << "Enter amount: ";

cin.ignore();

string strAmount;

getline(cin, strAmount);

bid.amount = strToDouble(strAmount, '$');

return bid;

}

/\*\*

\* Load a CSV file containing bids into a container

\*

\* @param csvPath the path to the CSV file to load

\* @return a container holding all the bids read

\*/

vector<Bid> loadBids(string csvPath) {

std::cout << "Loading CSV file " << csvPath << endl;

// Define a vector data structure to hold a collection of bids.

vector<Bid> bids;

// initialize the CSV Parser using the given path

csv::Parser file = csv::Parser(csvPath);

try {

// loop to read rows of a CSV file

for (int i = 0; i < file.rowCount(); i++) {

// Create a data structure and add to the collection of bids

Bid bid;

bid.bidId = file[i][1];

bid.title = file[i][0];

bid.fund = file[i][8];

bid.amount = strToDouble(file[i][4], '$');

//cout << "Item: " << bid.title << ", Fund: " << bid.fund << ", Amount: " << bid.amount << endl;

// push this bid to the end

bids.push\_back(bid);

}

} catch (csv::Error &e) {

std::cerr << e.what() << std::endl;

}

return bids;

}

// FIXME (2a): Implement the quick sort logic over bid.title

/\*\*

\* Partition the vector of bids into two parts, low and high

\*

\* @param bids Address of the vector<Bid> instance to be partitioned

\* @param begin Beginning index to partition

\* @param end Ending index to partition

\*/

int partition(vector<Bid>& bids, int begin, int end) {

//set low and high equal to begin and end

// pick the middle element as pivot point

// while not done

// keep incrementing low index while bids[low] < bids[pivot]

// keep decrementing high index while bids[pivot] < bids[high]

/\* If there are zero or one elements remaining,

all bids are partitioned. Return high \*/

// else swap the low and high bids (built in vector method)

// move low and high closer ++low, --high

//return high;

//local variable declaration

int midPoint;

string pivot;

Bid tempSwap;

bool done;

midPoint = begin + (end - begin) / 2;

pivot = bids[midPoint].title;

done = false;

while (!done) {

while (bids[begin].title < pivot) {

begin++;

}

while (pivot < bids[end].title) {

end--;

}

if (begin >= end) {

done = true;

}

else {

//set-up the swaps using a tempSwap variable of type Bid

tempSwap = bids[begin];

bids[begin] = bids[end];

bids[end] = tempSwap;

begin++;

end--;

}

}

return end;

}

/\*\*

\* Perform a quick sort on bid title

\* Average performance: O(n log(n))

\* Worst case performance O(n^2))

\*

\* @param bids address of the vector<Bid> instance to be sorted

\* @param begin the beginning index to sort on

\* @param end the ending index to sort on

\*/

void quickSort(vector<Bid>& bids, int begin, int end) {

//set mid equal to 0

/\* Base case: If there are 1 or zero bids to sort,

partition is already sorted otherwise if begin is greater

than or equal to end then return\*/

/\* Partition bids into low and high such that

midpoint is location of last element in low \*/

// recursively sort low partition (begin to mid)

// recursively sort high partition (mid+1 to end)

int lowIndex;

if (begin >= end) {

return;

}

//call the partition function to define the lowIndex

lowIndex = partition(bids, begin, end);

//recursive calls to quickSort

quickSort(bids, begin, lowIndex);

quickSort(bids, lowIndex + 1, end);

}

// FIXME (1a): Implement the selection sort logic over bid.title

/\*\*

\* Perform a selection sort on bid title

\* Average performance: O(n^2))

\* Worst case performance O(n^2))

\*

\* @param bid address of the vector<Bid>

\* instance to be sorted

\*/

void selectionSort(vector<Bid>& bids) {

//define min as int (index of the current minimum bid)

// check size of bids vector

// set size\_t platform-neutral result equal to bid.size()

// pos is the position within bids that divides sorted/unsorted

// for size\_t pos = 0 and less than size -1

// set min = pos

// loop over remaining elements to the right of position

// if this element's title is less than minimum title

// this element becomes the minimum

// swap the current minimum with smaller one found

// swap is a built in vector method

//local variable declaration

int indexSmallest = 0;

Bid tempSwap;

for (int i = 0; i < bids.size() - 1; i++) {

indexSmallest = i;

for (int j = i + 1; j < bids.size(); j++) {

if (bids[j].title < bids[indexSmallest].title) {

indexSmallest = j;

}

}

//swap the vector positions using a tempSwap variable of type Bid

tempSwap = bids[i];

bids[i] = bids[indexSmallest];

bids[indexSmallest] = tempSwap;

}

}

/\*\*

\* Simple C function to convert a string to a double

\* after stripping out unwanted char

\*

\* credit: http://stackoverflow.com/a/24875936

\*

\* @param ch The character to strip out

\*/

double strToDouble(string str, char ch) {

str.erase(remove(str.begin(), str.end(), ch), str.end());

return atof(str.c\_str());

}

/\*\*

\* The one and only main() method

\*/

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

// process command line arguments

string csvPath;

switch (argc) {

case 2:

csvPath = argv[1];

break;

default:

csvPath = "U:/CS 300 Vector Sorting Assignment Student Files/eBid\_Monthly\_Sales.csv"; //large set

//csvPath = "U:/CS 300 Vector Sorting Assignment Student Files/eBid\_Monthly\_Sales\_Dec\_2016.csv"; //small set

}

// Define a vector to hold all the bids

vector<Bid> bids;

// Define a timer variable

clock\_t ticks;

//add a few local variables for try/catch and usabiolity needs

const int GLOBAL\_SLEEP\_VALUE = 5000;

int choice = 0;

string anyKey = " ";

bool goodInput;

while (choice != 9) {

std::cout << "Menu:" << endl;

std::cout << " 1. Load Bids" << endl;

std::cout << " 2. Display All Bids" << endl;

std::cout << " 3. Selection Sort All Bids" << endl;

std::cout << " 4. Quick Sort All Bids" << endl;

std::cout << " 9. Exit" << endl;

std::cout << "Enter choice: ";

try { //add a try catch to protect against bad input

cin >> choice;

if ((choice > 0 && choice < 5) || (choice == 9)) {// limit the user menu inputs to good values

goodInput = true;

}

else {//throw error for catch

goodInput = false;

throw 1;

}

switch (choice) { //create a swtich to allow the menu to work

case 1:

// Initialize a timer variable before loading bids

ticks = clock();

// Complete the method call to load the bids

bids = loadBids(csvPath);

std::cout << bids.size() << " bids read" << endl;

// Calculate elapsed time and display result

ticks = clock() - ticks; // current clock ticks minus starting clock ticks

std::cout << "time: " << ticks << " clock ticks" << endl;

std::cout << "time: " << ticks \* 1.0 / CLOCKS\_PER\_SEC << " seconds" << endl;

Sleep(GLOBAL\_SLEEP\_VALUE);

break;

case 2:

// Loop and display the bids read

for (int i = 0; i < bids.size(); ++i) {

displayBid(bids[i]);

}

std::cout << "Press any key to continue...";

cin >> anyKey;

break;

case 3:

//selection sort switch

//start the clock with the tick variable and then call the function

//stop the clock with tick again and then outpout the time it tookto run

//sleep for some amount of time and then redraw the menu

ticks = clock();

selectionSort(bids);

// Calculate elapsed time and display result

ticks = clock() - ticks; // current clock ticks minus starting clock ticks

std::cout << "time: " << ticks << " clock ticks" << endl;

std::cout << "time: " << ticks \* 1.0 / CLOCKS\_PER\_SEC << " seconds" << endl;

Sleep(GLOBAL\_SLEEP\_VALUE);

break;

case 4:

//quick sort switch

//start the clock with the tick variable and then call the function

//stop the clock with tick again and then outpout the time it took to run

//sleep for some amount of time and then redraw the menu

ticks = clock();

quickSort(bids, 0, bids.size() - 1);

// Calculate elapsed time and display result

ticks = clock() - ticks; // current clock ticks minus starting clock ticks

std::cout << "time: " << ticks << " clock ticks" << endl;

std::cout << "time: " << ticks \* 1.0 / CLOCKS\_PER\_SEC << " seconds" << endl;

Sleep(GLOBAL\_SLEEP\_VALUE);

break;

case 9:

//default case for the exit statement so we don't fail the try catch

break;

default:

throw 2;

}

}

catch (int err) {

std::cout << "\nPlease check your input." << endl;

Sleep(GLOBAL\_SLEEP\_VALUE);

}

//need to clear the cin operator of extra input, e.g., 9 9, or any errors generated by bad input, e.g., 'a'

cin.clear();

cin.ignore();

//clear the consolse to redraw a fresh menu

system("cls");

}

std::cout << "Good bye." << endl;

Sleep(GLOBAL\_SLEEP\_VALUE);

return 0;

}