

Assignment 1 Report

Statistical Library

**Members**:

Đinh Duy Kha 1411675

Nguyễn Xuân Danh 1652736

Cù Huy Chiến 1752109

## Introduction

### Tasks

This assignment asks our group to build a statistical library that support data type **char**, **int** and **double**, having the following functions

1. Calculates sum
2. Calculate mean
3. Calculate standard deviation
4. Calculate variance
5. Calculate Q1, median and Q3
6. Calculate mode
7. Calculate and visualize histogram
8. Calculate and visualize noise

### Challenges

Statistics and probabilities is a rather new topic for us because we have not taken the course on this subject. To implement some functions of this project, we had to research on fundamental knowledge of statistic and probabilities.

### Requirements

1. Build a user interface support loading dataset of 3 types: **char**, **int** and **double**.
2. Write functions that are required for the library.

## Main tool used

### Code::Block

Code::Block is the basic IDE of taught in the course. We decided to use it in this project because it is simple and easy to use.

### Git and Github

Git is a version control system that is simple and fast. A version control system helps us synchronize our code across different computers. This helps us speed up the development process tremendously.

Github[[1]](#footnote-1) is a free-to-use Git repository hosting service. It is used by more than 31 million developers across many fields, with more than 100 million repositories hosted[[2]](#footnote-2).

The Github for our project is at: <https://github.com/dalo2903/statistical-lib>

## Implementation

### Classes

To increase the modularity of the code, we decided to split the program into 3 classes that do different tasks of the program. Those classes are:

#### statistical\_lib

This class contains the main operations to perform calculations on data. It supports up to 3 types of data: char, int and double by using the defined *data* union type. The class contains a vector of data as and contains an int variable *type* to store the current type of the dataset

The class contains the functions:

1. double sum() : Calculate sum of dataset.
2. double mean() : Calculate mean of dataset.
3. double standard\_deviation() : Calculate standard deviance of dataset.
4. double variance() : Calculate variance of dataset.
5. double Q1() : Calculate Q1 of dataset.
6. double median() : Calculate median of dataset.
7. double Q3() : Calculate Q3 of dataset.
8. void histogram() : Visualize histogram of dataset
9. void noise() : Reduce noise and visualize the dataset

#### data\_loader

This class implement methods to load data from files into memory. It mainly utilize **fopen()** and **fscanf()** functions of <*stdio.h*>. This class has the functions:

1. bool open\_file(string file\_name)

: Open a file using fopen(). Return true if success, false otherwise.

1. void load\_char\_data(string file\_name, vector<data> &d)

: Load data of type char from a file to a vector d.

1. void load\_int\_data(string file\_name, vector<data> &d)

: Load data of type int from a file to a vector d.

1. void load\_double\_data(string file\_name, vector<data> &d)

: Load data of type double from a file to a vector d.

#### menu\_printer

This class mainly contains functions to print out the user interface of the program. Those function are implemented mainly using *<iostream>* library for printing , <*iomanip*> and <*windows.h*> libraries for formatting. This class contains the following functions:

1. void print\_title() : Print the header of the program’s interface.
2. void print\_main\_menu\_no\_data() : Print main interface when there is no data loaded.
3. void print\_main\_menu() : Print main interface when there is no data loaded.
4. void print\_load\_data\_menu() : Print the interface of data loading process.

### Data types

#### union type

The *union* is used to store 3 type of data type: **int**, **char** and **double** inside a single variable.

### Data structures

#### std::vector

Vector is a type of data structure containing a list of data. Its implementation is similar to dynamic array in C.

#### std::map

Map stores element in a pair of key and value. It is useful when constructing a frequency table.

### Functions

#### Sum

**Sum** is just iterating over every element, then add them into a variable

double sum(){

double sum = 0;

for(vector<data>::iterator it = \_data.begin(); it != \_data.end(); ++it){

switch(type){

case TYPE\_CHAR:

sum += (\*it).c;

break;

case TYPE\_INT:

sum += (\*it).i;

break;

case TYPE\_DOUBLE:

sum += (\*it).d;

break;

}

}

return sum;

}

#### Mean

**Mean** is calculated by first calculating the sum, then divide it by the size of the dataset.

double mean(){

double mean = 0;

double sum = 0;

size\_t length = \_data.size();

for(vector<data>::iterator it = \_data.begin(); it != \_data.end(); ++it){

switch(type){

case TYPE\_CHAR:

sum += (\*it).c;

break;

case TYPE\_INT:

sum += (\*it).i;

break;

case TYPE\_DOUBLE:

sum += (\*it).d;

break;

}

}

mean = sum/length;

return mean;

}

#### Standard deviation and Variance

**Variance** is calculated by iterating over the dataset, first calculating the difference of each element and the mean of the dataset, square it, then add it to a sum. The sum is then divided by the size of the dataset and square rooted, resulting in variance.

**Standard deviation** is calculated the same way as variance, just without the last part with is taking square root.

double standard\_deviation(){

double standard\_deviation = 0;

double \_mean = mean();

size\_t length = \_data.size();

for(vector<data>::iterator it = \_data.begin(); it != \_data.end(); ++it){

switch(type){

case TYPE\_CHAR:

standard\_deviation += ((double)((\*it).c) - \_mean)\*((double)((\*it).c) - \_mean);

break;

case TYPE\_INT:

standard\_deviation += ((double)((\*it).i) - \_mean)\*((double)((\*it).i) - \_mean);

break;

case TYPE\_DOUBLE:

standard\_deviation += ((double)((\*it).d) - \_mean)\*((double)((\*it).d) - \_mean);

break;

}

}

standard\_deviation = sqrt(standard\_deviation/length);

return standard\_deviation;

}

double variance(){

double variance = 0;

double \_mean = mean();

size\_t length = \_data.size();

for(vector<data>::iterator it = \_data.begin(); it != \_data.end(); ++it){

switch(type){

case TYPE\_CHAR:

variance += ((double)((\*it).c) - \_mean)\*((double)((\*it).c) - \_mean);

break;

case TYPE\_INT:

variance += ((double)((\*it).i) - \_mean)\*((double)((\*it).i) - \_mean);

break;

case TYPE\_DOUBLE:

variance += ((double)((\*it).d) - \_mean)\*((double)((\*it).d) - \_mean);

break;

}

}

variance = variance /length;

return variance;

}

#### Q1, Median and Q3

**Median** can be divided into 2 cases. In the first case, the number of elements an odd number **n**, then the median would be the **(n + 1 )/ 2**th element. In the second case where the number of elements is an even number **n** , the median would be the average of the (**n / 2)**th number and the (**n / 2+1)**th .

double median(){

double median = 0;

int length = \_data.size();

if (length%2!=0) {

int index = (length + 1)/2;

switch (type) {

case TYPE\_CHAR:

median = \_data.at(index-1).c;

break;

case TYPE\_INT:

median = \_data.at(index-1).i;

break;

case TYPE\_DOUBLE:

median = \_data.at(index-1).d;

break;

}

}

else {

int index = length/2;

switch (type) {

case TYPE\_CHAR:

median = (double)(\_data.at(index-1).c + \_data.at(index).c)/2;

break;

case TYPE\_INT:

median = (double)(\_data.at(index-1).i + \_data.at(index).i)/2;

break;

case TYPE\_DOUBLE:

median = (double)(\_data.at(index-1).d + \_data.at(index).d)/2;

break;

}

}

return median;

}

**Q1** and **Q3** are calculated the same way as Median, with Q1 is the median of the first half of the dataset, and Q3 is the median of the second half of the dataset.

double Q1(){

double Q1 = 0;

int length = \_data.size();

int ele ;

if (length%2!=0) {

ele = (length -1)/2;

}

else {

ele = length/2;

}

if (ele%2!=0) {

int index = (ele + 1)/2;

switch (type) {

case TYPE\_CHAR:

Q1 = \_data.at(index-1).c;

break;

case TYPE\_INT:

Q1 = \_data.at(index-1).i;

break;

case TYPE\_DOUBLE:

Q1 = \_data.at(index-1).d;

break;

}

}

else {

int index = ele/2;

switch (type) {

case TYPE\_CHAR:

Q1 = (double)(\_data.at(index-1).c + \_data.at(index).c)/2;

break;

case TYPE\_INT:

Q1 = (double)(\_data.at(index-1).i + \_data.at(index).i)/2;

break;

case TYPE\_DOUBLE:

Q1 = (double)(\_data.at(index-1).d + \_data.at(index).d)/2;

break;

}

}

return Q1;

}

double Q3(){

double Q3 = 0;

int length = \_data.size();

int ele ;

if (length%2!=0) {

ele = (length -1)/2;

}

else {

ele = length/2;

}

if (ele%2!=0) {

int index = (ele + 1)/2 + ele +1;

switch (type) {

case TYPE\_CHAR:

Q3 = \_data.at(index-1).c;

break;

case TYPE\_INT:

Q3 = \_data.at(index-1).i;

break;

case TYPE\_DOUBLE:

Q3 = \_data.at(index-1).d;

break;

}

}

else {

int index = ele/2 + ele ;

switch (type) {

case TYPE\_CHAR:

Q3 = (double)(\_data.at(index-1).c + \_data.at(index).c)/2;

break;

case TYPE\_INT:

Q3 = (double)(\_data.at(index-1).i + \_data.at(index).i)/2;

break;

case TYPE\_DOUBLE:

Q3 = (double)(\_data.at(index-1).d + \_data.at(index).d)/2;

break;

}

}

return Q3;

}

#### Mode

We calculate **Mode** by first constructing a frequency table. The table is constructed using *std::map* , with the keys are unique elements of the dataset, and the values are their respective frequency. The largest frequency is taken, then modes are elements that have that frequency. The return value is a vector containing modes of the dataset.

vector<double> mode (){

vector<double> results;

map<char, int> frequency\_table\_char;

map<int, int> frequency\_table\_int;

map<double, int> frequency\_table\_double;

int \_max = 0;

switch(type){

case TYPE\_CHAR:

for(vector<data>::iterator it = \_data.begin(); it != \_data.end(); ++it){

frequency\_table\_char[(\*it).c]++;

}

for(map<char,int>::iterator it = frequency\_table\_char.begin(); it!=frequency\_table\_char.end(); ++it){

if(\_max < (\*it).second) \_max = (\*it).second;

}

for(map<char,int>::iterator it = frequency\_table\_char.begin(); it!=frequency\_table\_char.end(); ++it){

if(\_max == (\*it).second)

results.push\_back((\*it).first);

}

break;

case TYPE\_INT:

for(vector<data>::iterator it = \_data.begin(); it != \_data.end(); ++it){

frequency\_table\_int[(\*it).i]++;

}

for(map<int,int>::iterator it = frequency\_table\_int.begin(); it!=frequency\_table\_int.end(); ++it){

if(\_max < (\*it).second) \_max = (\*it).second;

}

for(map<int,int>::iterator it = frequency\_table\_int.begin(); it!=frequency\_table\_int.end(); ++it){

if(\_max == (\*it).second)

results.push\_back((\*it).first);

}

break;

case TYPE\_DOUBLE:

for(vector<data>::iterator it = \_data.begin(); it != \_data.end(); ++it){

frequency\_table\_double[(\*it).d]++;

}

for(map<double,int>::iterator it = frequency\_table\_double.begin(); it!=frequency\_table\_double.end(); ++it){

if(\_max < (\*it).second) \_max = (\*it).second;

}

for(map<double,int>::iterator it = frequency\_table\_double.begin(); it!=frequency\_table\_double.end(); ++it){

if(\_max == (\*it).second)

results.push\_back((\*it).first);

}

break;

}

return results;

}

#### Printing Histogram

To print the histogram, a frequency table first has to be constructed.

void histogram(){

SetConsoleTextAttribute(hConsole, 27);

cout<<"\t\tGenerated Histogram:"<<endl;

SetConsoleTextAttribute(hConsole, 15);

int max\_num = 0;

map<char, int> frequency\_table\_char;

map<int, int> frequency\_table\_int;

map<double, int> frequency\_table\_double;

switch(type){

case TYPE\_CHAR:

for(vector<data>::iterator it = \_data.begin(); it != \_data.end(); ++it){

frequency\_table\_char[(\*it).c]++;

}

for(map<char,int>::iterator it = frequency\_table\_char.begin(); it!=frequency\_table\_char.end(); ++it){

int length = (\*it).second ;

if(length> max\_num)

max\_num = length;

cout<<"\t"<<(\*it).first <<"\t|";

for (int i = 0; i< length; i++){

cout<<"\* ";

}

cout<<endl;

}

cout<<"\t"<<"\t"<<setw(max\_num\*3)<<setfill('-')<<""<<endl;

cout<<"\t"<<"\t ";

for(int i = 1; i<= max\_num; i++){

if(i>=10){

cout<<i;

}

else

cout<<i<<" ";

}

cout<<endl;

break;

case TYPE\_INT:

for(vector<data>::iterator it = \_data.begin(); it != \_data.end(); ++it){

frequency\_table\_int[(\*it).i]++;

}

for(map<int,int>::iterator it = frequency\_table\_int.begin(); it!=frequency\_table\_int.end(); ++it){

int length = (\*it).second ;

if(length> max\_num)

max\_num = length;

cout<<"\t"<<(\*it).first <<"\t|";

for (int i = 0; i< length; i++){

cout<<"\* ";

}

cout<<endl;

}

cout<<"\t"<<"\t"<<setw(max\_num\*3)<<setfill('-')<<""<<endl;

cout<<"\t"<<"\t ";

for(int i = 1; i<= max\_num; i++){

if(i>=10){

cout<<i;

}

else

cout<<i<<" ";

}

cout<<endl;

break;

case TYPE\_DOUBLE:

for(vector<data>::iterator it = \_data.begin(); it != \_data.end(); ++it){

frequency\_table\_double[(\*it).d]++;

}

for(map<double,int>::iterator it = frequency\_table\_double.begin(); it!=frequency\_table\_double.end(); ++it){

int length = (\*it).second ;

if(length> max\_num)

max\_num = length;

cout<<"\t"<<(\*it).first <<"\t|";

for (int i = 0; i< length; i++){

cout<<"\* ";

}

cout<<endl;

}

cout<<"\t"<<"\t"<<setw(max\_num\*3)<<setfill('-')<<""<<endl;

cout<<"\t"<<"\t ";

for(int i = 1; i<= max\_num; i++){

if(i>=10){

cout<<i;

}

else

cout<<i<<" ";

}

cout<<endl;

break;

}

}

#### Noise Reduction

Noise is detected using the z-score method. This method calculates the z-score for each sample with the formula: **z\_score = ( x – mean )/ standard deviation** . The z-score of each sample is then compare with a threshold, usually 3. If it is greater than the threshold, the sample would be considered noise.

void noise\_reduction(double threshold){

double standard\_deviation = this->standard\_deviation();

double mean = this->mean();

double z\_score = 0;

for(vector<data>::iterator it = \_data.begin(); it != \_data.end(); ){

switch(type){

case TYPE\_CHAR:

z\_score = (double)((\*it).c - mean )/ standard\_deviation;

if(z\_score > threshold){

cout<<"Noise detected :"<<(\*it).c;

it = \_data.erase(it);

}

else ++it;

break;

case TYPE\_INT:

z\_score = (double)((\*it).i - mean )/ standard\_deviation;

if(z\_score > threshold){

cout<<"Noise detected :"<<(\*it).i;

it = \_data.erase(it);

}

else ++it;

break;

case TYPE\_DOUBLE:

z\_score = (double)((\*it).d - mean )/ standard\_deviation;

if(z\_score > threshold){

cout<<"Noise detected :"<<(\*it).d;

it = \_data.erase(it);

}

else ++it;

break;

}

}

cout<<"Data after noise reduction: "<<endl;

histogram();

}

## Teamwork activities

**Day 1:** We decided to create the frame of our project. Kha and Danh decided to open up a project on Github so we can easily communicate and transfer codes of each person.

* **Kha** did the main frame, histogram and noise reduction and checking bug .
* **Danh** made the library and code from 1 to 4 including sum, mean, standard deviation and variance
* **Chien** code from 5 to 6, including the median, Q1, Q2 and mode.

**Day 2:** Team meeting and combining the code. Checking for bug and logical code. And start writing report.

**Day 3:** Checking the report and sum up the project. Then double check the project if there is any problem occurred during the checking process.

## Conclusion

After the project we are now able to understand and solving simple problems using C++. We got used to many functions and libraries as mentioned in our report. Also we have revised data structure and understood the basic of statistics. . Finally, we are able to find many pages that help us in the learning process.

## References

* <https://www.tutorialspoint.com/cplusplus/>
* <http://www.cplusplus.com/reference/>
* <https://www.mathsisfun.com/data/>
* <https://www.calculator.net/statistics-calculator.html>

1. <https://github.com> [↑](#footnote-ref-1)
2. <https://github.com/about> [↑](#footnote-ref-2)