/\*\* @file name: main.cpp

\* @brief description of file.

\* This file demonstrates the object creation.

\* The use of function callers

\* The process of encapsulation

\* The process of aggregation

\*Etc.

\*

\* Longer description of file.

\* This file has the main function, there are to objects one for the User Class and the other for the User Info class

\* There is a function call printing the above comment to meeting requirement LO4

\* OOP is a programming style based on the concept of objects which may contain data in the form of fields (Attributes) and code in the form of procedure (Methods)

\* Procedural programming is based on the concept of the procedure call also known as routines, subroutines, or functions. (Computational steps)

\* Functional programming style is based on the concept of evaluation of mathematical functions and avoid changing state and mutable data

\* There are other function demonstrating concepts like default and overload constructors

\* Encapsulation is also demonstrated in this file using setters and getters as well as aggregation concepts.

\* @author Name Sheimy Paz

\* @bug No known bugs.

\*

\*/

#include<iterator>

#include <iostream>

#include <ctime>

#include <vector>

#include <string>

#include "User.h"

#include "UserInfo.h"

#include "TransactionData.h"

using namespace std;

/\*\* @brief Short description of function.

\* Create objects User generalUser and UserInfo individualUsers

\* some callers

\*

\* @return The word void or a description of what is returned

\* it is returnning an integer

\*/

auto main()->int {

// Creating an objects of my base class User and subclass UserInfo to this will call the constructor

User generalUser;

UserInfo individualUsers;

//Callers

individualUsers.printComparison();

generalUser.myFunction();

generalUser.myFunction("\nAnd additionally", " Demostrate the concept of Costruction Overloading");

individualUsers.print();

//Encapsulation

individualUsers.set(5);

std::cout << "\n-----------------------------------------------------------------------------------------------------------------------\n";

std::cout << "This line of code shows encapsulation, The private varible x is set to "<< individualUsers.get()<<" and then it is access to it through a getter";

std::cout << "\n-----------------------------------------------------------------------------------------------------------------------";

//block chain code start here

Blockchain PazCoin;

//data for first block

TransactionData data1;

time\_t data1Time;

data1.amount = 1.5;

data1.receiverKey = "From Sher";

data1.senderKey = "To Ross.";

data1.timestamp = time(&data1Time);

std::cout << "\n\n-----------------------------------------------------------------------------------------------------------------------";

PazCoin.addBlock(data1);

//Using agregation

Address add1 = Address(data1.amount, data1.receiverKey, data1.senderKey, data1.timestamp = time(&data1Time));

Person p1 = Person(" Transfering", &add1);

p1.display();

//if the chain is valid we are getting 1

cout << "It's chain Valid?" << endl << PazCoin.isChainValid() << endl;

std::cout << "-----------------------------------------------------------------------------------------------------------------------";

//data for second block

TransactionData data2;

time\_t data2Time;

data2.amount = 5.9;

data2.receiverKey = "From David";

data2.senderKey = "To Paul.";

data2.timestamp = time(&data2Time);

std::cout << "\n\n-----------------------------------------------------------------------------------------------------------------------";

PazCoin.addBlock(data2);

//Using agregation

Address add2 = Address(data2.amount, data2.receiverKey, data2.senderKey, data2.timestamp = time(&data2Time));

Person p2 = Person(" Transfering", &add2);

p2.display();

//if the chain is valid we are getting 1

cout << "It's chain Valid?" << endl << PazCoin.isChainValid() << endl;

std::cout << "-----------------------------------------------------------------------------------------------------------------------\n";

//trying to manipulate data-> get sneaky

Block\* hackBlock = PazCoin.getLatestBlock();

hackBlock->data.amount = 1000;

hackBlock->data.receiverKey = "thief";

//if the chain is not valid we are getting 0

//Using agregation

Address add3 = Address(hackBlock->data.amount, hackBlock->data.receiverKey, "Triying to steal. Curent Time Stamp:", data1.timestamp = time(&data1Time));

Person thief = Person(" Transfering", &add3);

thief.display();

cout << "It's chain Valid?" << endl << PazCoin.isChainValid() << endl;

std::cout << "-----------------------------------------------------------------------------------------------------------------------\n";

return 0;

}

/\*\* @File name:UserInfo.h

\* @brief Brief description of file.

\* this file demonstrates the use of subclassing the design of a simple class.

\* Hierarchies allowing code to be reuse.

\* Subclass User Info from User base class.

\*

\*

\* Longer description of file.

\* It prints information to screen.

\* Some explanation about the project and its purpose.

\* Has some string attributes explaining the comparison between OOP PP and FP.

\* Uses subclassing.

\* Demonstrate the concept of constructors.

\* Has an int private attribute used to demonstrate encapsulation.

\* Has comments describing visibility on inheritance models.

\* Has comments describing relationship between object-oriented inheritance and subtyping.

\*

\*

\* @author name: Sheimy Paz

\* @bug No known bugs.

\*/

#pragma once

#include "User.h"

using namespace std;

/\*\* @brief Short description of function.

\* Clase UserInfo which is derived fron base class User

\* Has a private member x user to demonstrate encapsulation

\*

\* Longer description of function.

\*

\* @param One for each parameter with the name and a description

\* N/A

\* @return The word void or a description of what is returned

\* none

\*/

//LO2.Use subclassing to design simple class hierarchies that allow code to be reused for distinct subclasses.

class UserInfo :

public User

{

private:

// data hidden from outside world

int x =0;

public:

//LO4.Include a comment in which you compareand contrast the procedural / functional approach and the object - oriented approach

string welcome = "In this project we are going to see implementation of key learning skills form Object oriented programing class.\nAlong with that there is a small and very basic block chain implementation with the object of understand how it works.\n";

string opp = "OOP is a programming style based on the concept of objects which may contain data in the form of fields(Attributes) and code in the form of procedure (Methods).\n";

string pp = "Procedural programming is based on the concept of the procedure call also known as routines, subroutines, or functions (Computational steps).\n";

string fp = "Functional programming style is based on the concept of evaluation of mathematical functions and avoid changing state and mutable data.\n";

// constructor is a special method that is automatically called when an object of a class is created.

//To create a constructor, use the same name as the class, followed by parentheses ():

UserInfo() {}

void printComparison() {

cout << "\n-----------------------------------------------------------------------------------------------------------------------\n";

cout << welcome;

cout << "-----------------------------------------------------------------------------------------------------------------------\n";

cout << "\n-----------------------------------------------------------------------------------------------------------------------\n";

cout<< "Comparison between Procedural/Funtional and OOP approach: \n\n" << opp << pp << fp;

cout << "-----------------------------------------------------------------------------------------------------------------------\n";

}

/\*LO2a.Include comments describing the visibility inheritance model

\* When a base class is derived by a derived class with the help of inheritance, the accessibility of base class by the derived class is controlled by visibility modes.

\* The derived class doesn’t inherit access to private data members.However, it does inherit a full parent object, which contains any private members which that class declares.

\* for example:

\* A public member visibility will be open to all.It means that any class can accessand use this members.

\* A protected member visibility will be only to the derived class.It means that any derived class can accessand use any protected member.

\* A private member visibility will not be open to any other class.It means that any derived class cannot access or use its members.\*/

/\*LO5.Explain the relationship between object - oriented inheritance(code - sharing and overriding) and subtyping(the idea of a subtype being usable in a context that expects the supertype).

\* subtyping(also subtype polymorphism or inclusion polymorphism) is a form of type polymorphism in which a subtype is a datatype that is related to another datatype(the supertype)

\* by some notion of substitutability, meaning that program elements, typically subroutines or functions, written to operate on elements of the supertype can also operate on elements of the subtype.

\* In object-oriented programming, inheritance is the mechanism of basing an object or class upon another object (prototype-based inheritance) or class (class-based inheritance), retaining similar

\* implementation. Also defined as deriving new classes (sub classes) from existing ones such as super class or base class and then forming them into a hierarchy of classes. In most class-based

\* object-oriented languages, an object created through inheritance, a "child object", acquires all the properties and behaviors of the "parent object", with the exception of: constructors, destructor,

\* overloaded \*operators and friend functions of the base class.

\*/

//function to set value of

// variable x

void set(int a)

{

x = a;

}

// function to return value of

// variable x

int get()

{

return x;

}

void print() {

cout << "\n-----------------------------------------------------------------------------------------------------------------------\n";

cout << "\n-----------------------------------------------------------------------------------------------------------------------\n";

cout << x;

cout << "This line of code represent the use of a sub class and describe the visibility inheritance model.\n";

cout << "A public member visibility will be open to all.It means that any class can accessand use this members.\nA protected member visibility will be only to the derived class. It means that any derived class can accessand use any protected member.\nA private member visibility will not be open to any other class.It means that any derived class cannot access or use its members.\n";

cout << "-----------------------------------------------------------------------------------------------------------------------\n";

}

protected:

};

/\*\* @file name: UserInfo.cpp

\*

\* @brief description of file.

\* This file as comment satisfiying requirement LO3

\*

\*

\* Longer description of file.

\* The comment explain the Correctly reason about control flow in a program using dynamic dispatch.

\* @author Name Sheimy Paz

\* @bug No known bugs.

\*/

#include "UserInfo.h"

//dynamic dispatch is the process of selecting which implementation of a polymorphic operation (method or function) to call at run time

/\*\* @File name: User.h

\* @brief Brief description of file.

\*this file demonstrate the disign of a simple class named User

\*which is reused to make class named: UserInfo using hierarchies

\* Longer description of file.

\* it print information to screen

\*

\* @author name: Sheimy Paz

\* @bug No known bugs.

\*/

#pragma once

#include <iostream>

#include <ctime>

#include <vector>

#include <string>

//LO1.Designand implement a class.

//LO1a. At least one class in a header file with non-trivial methods implemented in a cpp file

//Class name: User

class User{

private:

public:

std::string name{};

std::string a = "a";

std::string b = "b";

//LO1b.Overload a constructor

void myFunction();

void myFunction(std::string a, std::string b);

void myFunction(std::string a);//function declaration

//LO1c.Utilize an initialization list

User() {}

User(std::string name) : name("ale")

{

}

protected:

};

class Address

{

public:

double amountC;

std::string sender, receiver;

time\_t timeStamp = 0;

Address(double amountC, std::string sender, std::string receiver, time\_t timeStamp)

{

this->amountC = amountC;

this->sender = sender;

this->receiver = receiver;

this->timeStamp = timeStamp;

}

};

class Person

{

private:

Address\* address;

public:

std::string name;

Person(std::string name, Address\* address)

{

this->name = name;

this->address = address;

}

void display()

{

std::cout << name << " " << " " << address->amountC << " " << address->sender << " " << address->receiver << " " << address->timeStamp << std::endl;

}

};

/\*\* @file name: User.cpp

\*

\* @brief description of file.

\* This file as a header file with a non-trivial method implemented

\* A comment meeting requirement LO1a

\* It demonstrate construction overloading

\*

\*

\* Longer description of file.

\* No necessary

\* @author Name Sheimy Paz

\* @bug No known bugs.

\*/

#include "User.h"

//LO1a.At least one class in a header file with non - trivial methods implemented in a cpp file

void User::myFunction() {

std::cout << "\n-----------------------------------------------------------------------------------------------------------------------\n";

std::cout << "This part of the code demostrate the content of the base class in a header file being implementd on a cpp file." << name; }

void User::myFunction(std::string a, std::string b) {

std::cout << a << b;

}

void User::myFunction(std::string a) {}

/\*\* @file TransactionData.h

\* @brief Brief description of file.

\* This file demonstrate structure concepts

\* It has classes, Functions and constructions that provide de creation of a geneses block, couple of blocks and a blockchain

\* That are being used to create transaction information.

\* Demontrate the use of an iterator

\*

\* Longer description of file.

\* In this case the structure has a double, two strings and a time\_t member

\* this conform a couple transaction of transaction that are being verify as valid or not valid using

\* a hash and a previous hash with an iterator that confirm the block with a one if is valid or with

\* 0 if is not valid

\*

\*

\* @author Name: Sheimy Paz

\* @bug No known bugs.

\*/

#pragma once

#ifndef TransactionData\_h

#define TransactionData\_h

#include <string>

#include <ctime>

#include <vector>

using namespace std;

/\*\* @brief Short description of function.

\* struct TransactionData initialize attributes

\* That are being used to create block for the blockchain

\* It has two constructors one default and the other one

\* Initializing it attributes

\*

\*Default constructor does not take any parameter

\*TransactionData()noexcept {};

\*Overloaded constructor take 4 parameters

\* TransactionData(double amt, std::string sender, std::string receiver, time\_t time)

\* Double, string, string, and time\_t

\*

\* Double amount @param is one of the attributes and it is used to specify the number of coins per user transaction

\* string senderKey @param is one of the attribute and it is used to define for the user sending the transaction

\* string receiverKey @param is one of the attribute and it is used to define for the user receiving the transaction

\* time\_t timestamp @param is one of the attribute and it is used to define timestamo for each transaction

\* @return the word void or a description of what is returned

\* No return

\*/

struct TransactionData

{

double amount{ 0 };

std::string senderKey;

std::string receiverKey;

time\_t timestamp{ 0 };

TransactionData()noexcept {};

TransactionData(double amt, std::string sender, std::string receiver, time\_t time)

{

amount = amt;

senderKey = sender;

receiverKey = receiver;

timestamp = time;

};

};

/\*\* @brief Short description of function.

\* Class block has 4 private atributes

\* has some getters and setters to access this atributes

\* and and some constructors

\*

\*/

class Block

{

private:

int index;

size\_t blockHash;

size\_t previousHash;

auto generateHash()->size\_t;

public:

//LO1b->Overload a constructor

Block(int indx, TransactionData data);

// Constuctor

Block(int idx, TransactionData d, size\_t prevHash);

// Get Index

auto getIndex()->int;

// Get Original Hash

auto getHash()->size\_t;

// Get Previous Hash

auto getPreviousHash()->size\_t;

// Transaction Data

// ideally will be a private member with a "getter": getData()

TransactionData data;

// Validate Hash

auto isHashValid()->bool;

};

inline Block::Block(int indx, TransactionData data)

{

}

// Constructor with params

Block::Block(int idx, TransactionData d, size\_t prevHash)

{

index = idx;

data = d;

previousHash = prevHash;

blockHash = generateHash();

}

//Generates hash for current block

//- Includes previousHash in generation

auto Block::generateHash()->size\_t

{

//part1

hash<string> hash1; // hashes transaction data string

hash<size\_t> hash2;

hash<size\_t> finalHash;

// creating string of transaction data

string toHash = to\_string(data.amount) + data.receiverKey + data.senderKey + to\_string(data.timestamp);

return finalHash(hash1(toHash) + hash2(previousHash));

}

// Public Functions

auto Block::getHash()->size\_t { return blockHash; }

// Public Functions

auto Block::getPreviousHash()->size\_t { return previousHash; }

// Public Functions

//making them equal will mada the function invalid if someone manipulated any transaction data and tried to generate the block again cause the hash willno //be equal again

auto Block::isHashValid()->bool { return generateHash() == getHash(); }

class Blockchain

{

private:

auto createGenesisBlock()->Block;

public:

std::vector<Block> chain;

// Constuctor

Blockchain();

// Public Functions

auto getChain()->std::vector<Block>;

auto getLatestBlock()->Block\*;

auto isChainValid() ->bool;

void addBlock(TransactionData data);

//void printChain();

//auto getIndex()->int;

};

//block chain class members

//constructor

//default not take any arguments is going to create the genesis block

Blockchain::Blockchain()

{

Block genesis = createGenesisBlock();

chain.push\_back(genesis); //push genesis block into block chain

}

//first block of the chain only need some data like current time and transaction data

// Genesis Block

auto Blockchain::createGenesisBlock()-> Block

{

//part 1

time\_t current = 0;

TransactionData d;

d.amount = 0;

d.receiverKey = "none";

d.senderKey = "none";

d.timestamp = time(&current);//pointer to current time

hash<int>hash1;

// Return Genesis Block

Block genesis(0, d, hash1(0));

return genesis;

}

inline auto Blockchain::getChain() -> std::vector<Block>

{

return std::vector<Block>();

}

//only for demo

//returning pointer

//pointer reference something in memory, it give us a memory addres its a pointer to a memory address

// with this we can manipulate the data inside that memory address

// if we were to do this without a pointer it will return the latest block of the chain but it would be immutable

// since we are returning a memory address wecan manipulate the data inside that address

//

Block\* Blockchain::getLatestBlock() { return &chain.back(); }

void Blockchain::addBlock(TransactionData d) {

int index = (int)chain.size() - 1;

Block newBlock(index, d, getLatestBlock()->getHash());

}

auto Blockchain::isChainValid()->bool

{

//part1

vector<Block>::iterator it; //itereate over a vector of type block

int chainLen = (int)chain.size();

for (it = chain.begin(); it != chain.end(); ++it) {

//if a currecnt block does not have a valid hash is return false

Block currentBlock = \*it;

if (!currentBlock.isHashValid())

{

return false;

}

if (chainLen > 1) {

Block previousBlock = \*(it - 1);

if (currentBlock.getPreviousHash() != previousBlock.getHash())

{

return false;

}

}

}

return true;

}

#endif /\* TransactionData\_h \*/